Decoder Trimming: A Feasible Alternative to Fine-Tuning for Text Classification with Large Language Models

SYNOPSIS SUBMITTED TO ASIAN SCHOOL OF MEDIA STUDIES IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE AWARD OF DEGREE OF

B. Sc.
in
Data Science

By

NIRMAL THOMAS

(University Roll No: 12112936004)

Under the Supervision of

Dr. Aashima Bangia

ASIAN SCHOOL OF MEDIA STUDIES
NOIDA

2024

DECLARATION

I, **NIRMAL THOMAS**, S/O **THOMAS P.A,** declare that my project entitled **Decode Trimming: A feasible alternative to Fine-Tuning for Text Classification with Large Language Models** submitted at **School of Data Science, Asian School of Media Studies, Film City, Noida**, for the award of **B.Sc. in Data Science, Noida International University** is an original work and no similar work has been done in India anywhere else to the best of my knowledge and belief.

This project has not been previously submitted for any other degree of this or any other University/Institute.

Signature

Nirmal Thomas
9711357759
Nirmal.thomas013@gmail.com
B.Sc. Data Science
School of Data Science
Asian School of Media Studies

ACKNOWLEDGEMENT

The completion of the project titled **Decode Trimming: A feasible alternative to Fine-Tuning for Text Classification with Large Language Models**, gives me an opportunity to convey my gratitude to all those who helped to complete this project successfully. I express special thanks:

- To ***Prof. Sandeep Marwah,*** President, Asian School of Media Studies, who has been a source of perpetual inspiration throughout this project.
- To ***Mr. Ashish Garg,*** Director for School of Data Science for your valuable guidance, support, consistent encouragement, advice and timely suggestions.
- To **Dr. Aashima Bangia**, Assistant Professor of School of Data Science, for your encouragement and support. I deeply value your guidance.
- To my ***friends*** for their insightful comments on early drafts and for being my worst critic. You are all the light that shows me the way.

  To all the people who have directly or indirectly contributed to the writing of this thesis, but their names have not been mentioned here.

Signature

Nirmal Thomas
9711357759
Nirmal.thomas013@gmail.com
B.Sc. Data Science
School of Data Science
Asian School of Media Studies

ABSTRACT

Recent releases of pre-trained Large Language Models (LLMs) have gained considerable traction and achieved impressive results in various natural language processing (NLP) tasks, including text classification. However, a major hurdle remains: fine-tuning, the prevalent method for adapting LLMs to specific tasks, suffers from high computational cost and substantial labeled data demands.

This study proposes decoder trimming, a novel alternative to fine-tuning for text classification with LLMs. While fine-tuning involves adapting the entire pre-trained model to a new task by updating all its parameters on a specific dataset, decoder trimming takes a more targeted approach.

Specifically, decoder trimming focuses on a critical bottleneck in LLMs: the last embedding decoder layer. This layer typically maps the model's internal understanding of the text (hidden states) into the vast vocabulary it recognizes. Decoder trimming proposes creating a new, reduced version of this layer. The new layer maintains the original input features but restricts its output to a subset of relevant tokens specifically tailored for the text classification task at hand. This approach aims to improve efficiency in two ways:

- **Reduced Model Size:** By focusing on a smaller vocabulary subset, the overall size of the model is significantly reduced, leading to faster training times and lower computational resource requirements.
- **Potentially Lower Data Demands:** The reduced complexity of the decoder layer might enable the model to achieve good performance with less training data compared to traditional fine-tuning.

We test this approach with the Gemma-2B variant model. To evaluate its effectiveness, we benchmark decoder trimming against state-of-the-the-art Machine Learning and Deep Learning models on established text classification datasets. Our findings are promising, demonstrating that decoder trimming presents a compelling and practical alternative. Not only does it achieve competitive accuracy scores (90.6 on AGNews Classification Benchmark and 94.13 on SST2 Benchmark), but remarkably, the Gemma-2b-Trimmed model surpasses current best performance (SOTA) on both benchmarks.

This research suggests that decoder trimming has the potential to significantly improve the accessibility and practicality of using LLMs for text classification tasks. By reducing computational costs and potentially lowering data requirements, decoder trimming opens doors for wider adoption of LLMs, even in scenarios with limited resources. Future work can explore the applicability of decoder trimming to different LLM architectures and investigate its effectiveness on more diverse text classification tasks.

TABLE OF CONTENTS

List of Figures

List of Tables

Abbreviation

LLM – Large Language Model

NLP – Natural Language Processing

BERT – Bi-directional Encoder Representation from Transformers

GPT – Generative Pretrained Transformers

AI – Artificial Intelligence

GPU – Graphical Processing Unit

ReLU – Rectified Linear Unit

MoE – Mixture of Experts

GenAI – Generative Artificial Intelligence

BPE – Byte Pair Encoding

RNN – Recurrent Neural Network

GELU – Gaussian Error Linear Unit

SWiGLU – Swish Gated Linear Unit

1. INTRODUCTION

The world of AI has witnessed a revolution in recent years, driven by the emergence of Large Language Models (LLMs). These powerful algorithms, trained on massive datasets of text and code, are fundamentally altering how we interact with machines and shaping the landscape of technology as a whole. LLMs possess an unprecedented ability to understand and generate human language. They can translate languages, write different kinds of creative content, and answer your questions in an informative way, blurring the lines between machine and human communication. This has led to a surge in applications across various sectors. However, this transformative power comes at a cost. Building and running LLMs requires immense computational resources. The training process consumes vast amounts of data and energy, making them expensive to develop and maintain. This cost barrier can limit access and create a scenario where only large corporations can leverage the full potential of these models. Despite these challenges, the potential benefits of LLMs are undeniable. As research continues and technology advances, we can expect to see these models become more efficient and accessible, further transforming the way we live, work, and interact with the world around us.

To unlock the true potential of Large Language Models (LLMs), there are two key techniques: Pre-Training and Fine-Tuning. These methods work together to bridge the gap between a model's raw language ability and its mastery of specific tasks. Pre-training is the initial step where this slate is filled with vast amounts of text data. This data can be anything from books and articles to code repositories and online conversations. The LLM devours this information, statistically identifying patterns and relationships between words and phrases. It essentially learns the building blocks of language – how words connect, how sentences are formed, and the nuances of meaning. Pre-training equips the LLM with a foundational understanding of language as a whole. This process is unsupervised, meaning the data isn't categorized or labeled for specific tasks. While pre-training provides a strong language foundation, LLMs aren't ready for specific tasks like writing poetry or summarizing news articles. Fine-tuning is basically to upskill a general language model for a focused task. We introduce a focused dataset of labeled examples relevant to the desired task. For example, to train for sentiment analysis, the data might consist of sentences labeled as positive, negative, or neutral. The LLM leverages its pre-trained knowledge as a base and adjusts its internal parameters based on the new, task-specific data. It hones its ability to identify sentiment cues within sentences, becoming an expert in that particular domain. Fine-tuning a pre-trained LLM is

significantly faster and less resource-intensive than training a new model from scratch. The LLM already has a strong grasp of language, so it only needs to adapt to the specifics of the task. It improves the LLM's performance in that specific area it is trained. The model learns the nuances and patterns relevant to the task, leading to more accurate and relevant outputs.

Another important distinction between contemporary LLMs and earlier NLP models is that earlier NLP models were typically designed for specific tasks, such as text categorization, language translation, etc. Whereas those earlier NLP models excelled in their narrow applications, LLMs demonstrate a broader proficiency across a wide range of NLP tasks.

The success behind LLMs can be attributed to the transformer architecture that underpins many LLMs and the vast amounts of data on which LLMs are trained, allowing them to capture a wide variety of linguistic nuances, contexts, and patterns that would be challenging to encode manually.

The idea of LLMs was first floated with the creation of Eliza in the 1960s: it was the world's first chatbot, designed by MIT researcher Joseph Weizenbaum. Eliza marked the beginning of research into natural language processing (NLP), providing the foundation for future, more complex LLMs.

Then almost 30+ years later, in 1997, Long Short-Term Memory (LSTM) networks came into existence. Their advent resulted in deeper and more complex neural networks that could handle greater amounts of data. Stanford's CoreNLP suite, introduced in 2010, was the next stage of growth allowing developers to perform sentiment analysis and named entity recognition.

Subsequently, in 2011, a smaller version of Google Brain appeared with advanced features such as word embeddings, which enabled NLP systems to gain a clearer understanding of context. This was a significant turning point, with transformer models bursting onto the scene in 2017. Think GPT, which stands for Generative Pre-trained Transformer, has the ability to generate or "decode" new text. Another example is BERT - Bidirectional Encoder Representations from Transformers. BERT can predict or classify input text based on encoder components.

From 2018 onward, researchers focused on building increasingly larger models. It was in 2019 that researchers from Google introduced BERT, the two-directional, 340-million parameter model (the third largest model of its kind) that could determine context allowing it to adapt to various tasks. By pre-training BERT on a

wide variety of unstructured data via self-supervised learning, the model was able to understand the relationships between words. In no time at all, BERT became the go-to tool for natural language processing tasks. In fact, it was BERT that was behind every English-based query administered via Google Search.

As BERT was becoming more refined, OpenAI's GPT-2, at 1.5 billion parameters, successfully produced convincing prose. Then, in 2020, they released GPT-3 at 175 billion parameters, which set the standard for LLMs and formed the basis of ChatGPT. It was with the release of ChatGPT in November 2022 that the general public really began to take notice of the impact of LLMs. Even non-technical users could prompt the LLM, receive a rapid response, and carry on a conversation, causing a stir of both excitement and foreboding.

Chiefly, it was the transformer model with an encoder-decoder architecture that spurred the creation of larger and more complex LLMs, serving as the catalyst for Open AI's GPT-3, ChatGPT, and more. Leveraging two key components: word embeddings (which allow a model to understand words within context) and attention mechanisms (which allow a model to assess the importance of words or phrases), transformers have been particularly helpful in determining context. They've been revolutionizing the field ever since due to their ability to process large amounts of data in one go.

Most recently, OpenAI introduced GPT-4, which is estimated at one trillion parameters — five times the size of GPT-3 and approximately 3,000 times the size of BERT when it first came out.

As we briefly mentioned earlier, studies have shown that larger language models that are programming languages with more parameters have better performance, thus spurring a race among developers and AI researchers to create bigger and better LLMs — the more parameters, the better!

The original GPT, which only had a few million parameters, morphed into models like BERT and GPT-2, comprising hundreds of millions of parameters. More recent examples include GPT-3, which has 175 billion parameters, while Megatron-Turing's language model has already exceeded 500 billion parameters (at 530 billion) — that's approximately two times larger for every three and a half months over the last four years!

However, AI lab DeepMind's RETRO (Retrieval-Enhanced Transformer) has demonstrated that it can outperform other existing models that are 25 times its size.

This is an elegant solution to the notable downsides of training larger models, as they typically require more time, resources, money, and computational energy to train. Not only that, but RETRO has demonstrated its large model has the potential to reduce harmful and toxic information via enhanced filtration capabilities. So, bigger isn't always better! The training process for the next enormous LLM could likely require all the text and training data available online, and smaller models that are more optimally trained could be the solution.

Furthermore, not long after the release of GPT-4, OpenAI's CEO Sam Altman stated that he believed the age of giant models had reached a point of no return given the limited number of data centers and information on the web. Many researchers now agree that via customization, smaller LLMs can be just as effective as large models, if not more so.

Along with the incredible advancements in AI, machine learning models, and LLMs on the whole, there is still no shortage of challenges to overcome. Misinformation, malware, discriminatory content, plagiarism, and information that is simply untrue can lead to unintended or dangerous outcomes; this calls these models into question.

What's more, when biases are inadvertently introduced into LLM-based products like GPT-4, they can come across as "certain but wrong" on some subjects. It's a bit like when you hear a politician talking about something they don't know anything about. Overcoming these limitations is key to building public trust with this new technology.

2. Dataset Description

In this study we have used two datasets to evaluate the proposed method against other SOTA model architectures.

I. Stanford Sentiment Treebank

The Stanford Sentiment Treebank is a corpus with fully labeled parse trees that allows for a complete analysis of the compositional effects of sentiment in language. The corpus is based on the dataset introduced by Pang and Lee (2005) and consists of 11,855 single sentences extracted from movie reviews. It was parsed with the Stanford parser and includes a total of 215,154 unique phrases from those parse trees, each annotated by 3 human judges.

Binary classification experiments on full sentences (negative or somewhat negative vs somewhat positive or positive with neutral sentences discarded) refer to the dataset as SST-2 or SST binary.

	Train	Validation	Test
Number of Examples	67349	872	1821

Table 2.1 Partition of SST2 Dataset

The Data set consists of the following columns:

a. idx: Monotonically increasing index ID.
b. sentence: Complete sentence expressing an opinion about a film.
c. label: Sentiment of the opinion, either "negative" (0) or positive (1). The test set labels are hidden (-1).

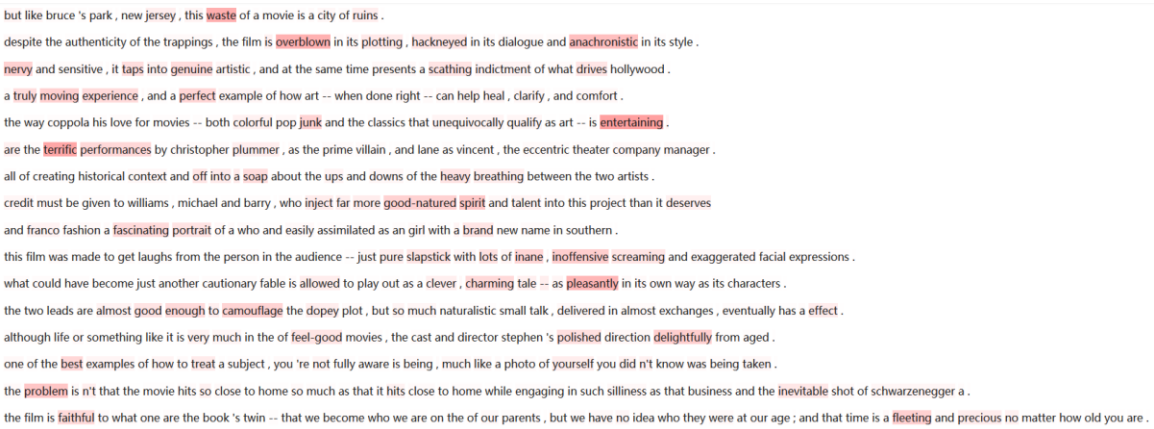


Fig 2.1 example of words in sentences that contributes to the sentiment of this sentence.

idx	sentence	label
0	it 's a charming and often affecting journey .	1
1	unflinchingly bleak and desperate	0
2	allows us to hope that nolan is poised to emba...	1
3	the acting , costumes , music , cinematography...	1
4	it 's slow -- very , very slow .	0

Figure 2.2: First five rows of the dataset

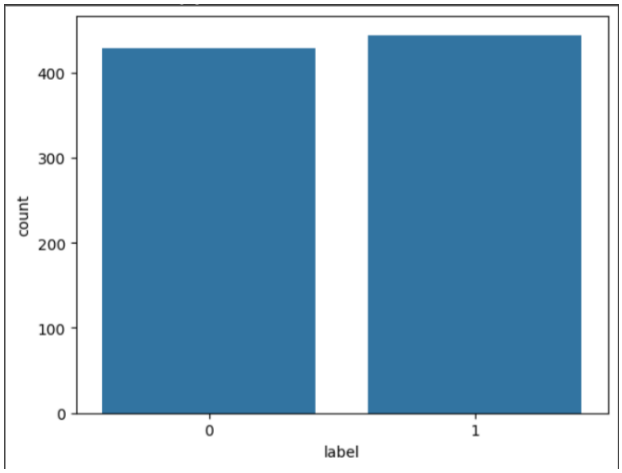


Figure 2.3: Count of Labels in the dataset (0: Negative , 1:Positive)

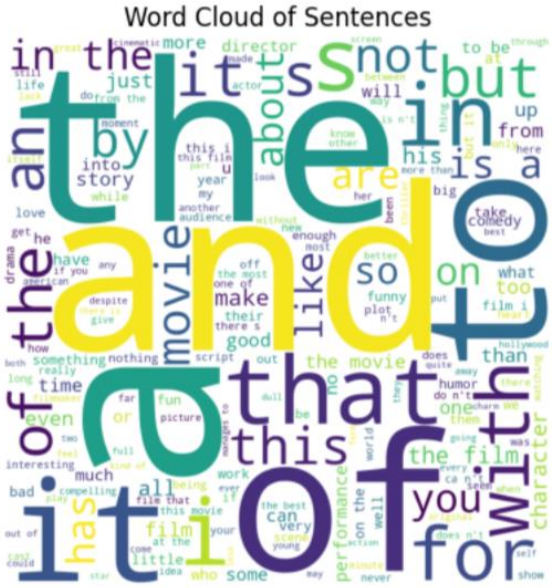


Figure 2.4: Word Cloud of SST-2 Dataset

AG’s News Topic Classification Dataset

AG is a collection of more than 1 million news articles. News articles have been gathered from more than 2000 news sources by ComeToMyHead in more than 1 year of activity. ComeToMyHead is an academic news search engine which has been running since July, 2004. The dataset is provided by the academic comunity for research purposes in data mining (clustering, classification, etc), information retrieval (ranking, search, etc), xml, data compression, data streaming, and any other non-commercial activity.

The AG's news topic classification dataset is constructed by choosing 4 largest classes from the original corpus. Each class contains 30,000 training samples and 1,900 testing samples. The total number of training samples is 120,000 and testing 7,600.

The Data set consists of the following columns:

d. label: class index where 1 is World, 2 is Sports, 3 is business and 4 is Sci/Tech
e. title: Heading about the news article
f. description: The text information about news

The title and description are escaped using double quotes ("), and any internal double quote is escaped by 2 double quotes (""). New lines are escaped by a backslash followed with an "n" character, that is "\n".

label	title	description
3	Fears for T N pension after talks	Unions representing workers at Turner Newall...
4	The Race is On: Second Private Team Sets Launc...	SPACE.com - TORONTO, Canada -- A second\team o...
4	Ky. Company Wins Grant to Study Peptides (AP)	AP - A company founded by a chemistry research...
4	Prediction Unit Helps Forecast Wildfires (AP)	AP - It's barely dawn when Mike Fitzpatrick st...
4	Calif. Aims to Limit Farm-Related Smog (AP)	AP - Southern California's smog-fighting agenc...

Figure 2.5: First 5 rows of the dataset

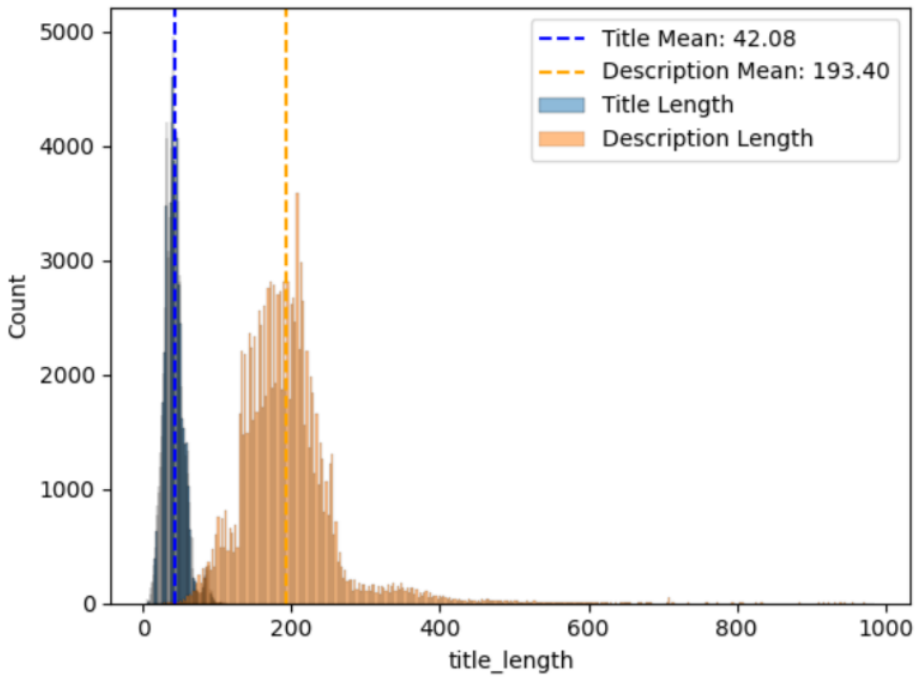


Figure 2.6: Text length analysis

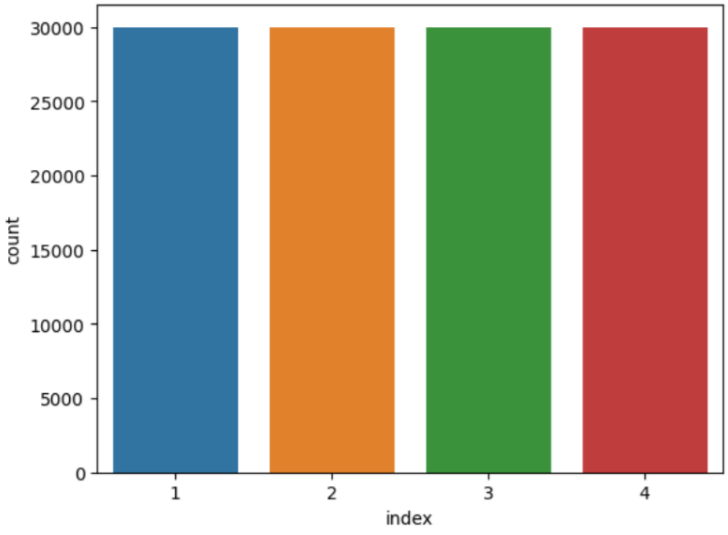


Figure 2.7: Class distribution
(1: World, 2: Sports, 3: Business, 4: Sci/Tech)

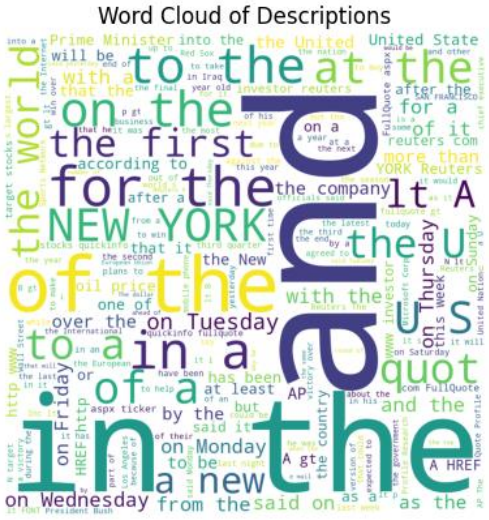
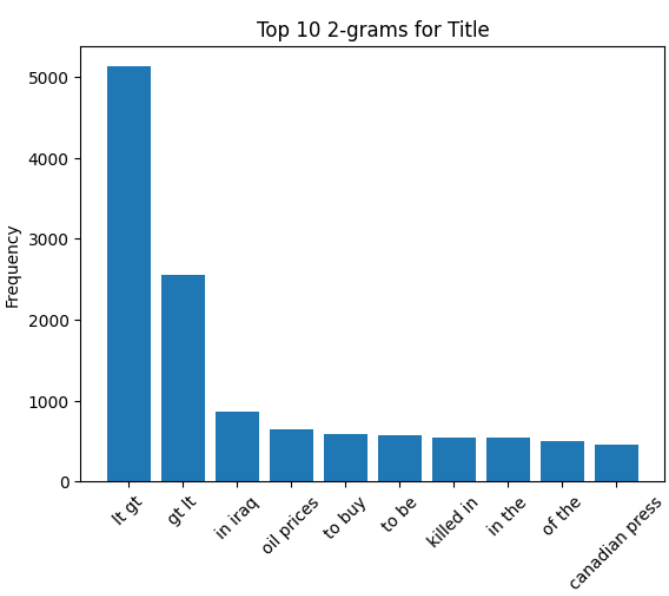


Figure 2.8: Word Cloud Analysis



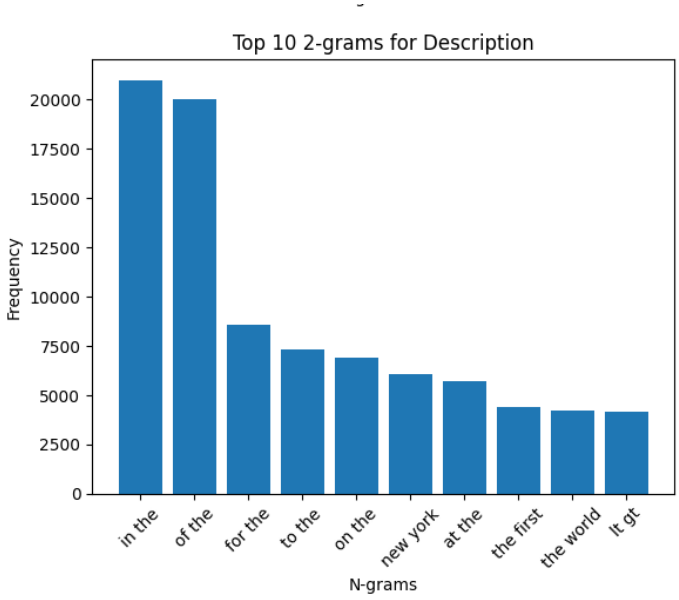


Figure 2.8: n-gram countplot

3. PROBLEM STATEMENT

Large language models (LLMs), such as those offered in OpenAI's ChatGPT, are deep neural network models that have been developed over the past few years. They ushered in a new era for natural language processing (NLP). Before the advent of LLMs, traditional methods excelled at categorization tasks such as email spam classification and straightforward pattern recognition that could be captured with handcrafted rules or simpler models. However, they typically underperformed in language tasks that demanded complex understanding and generation abilities, such as parsing detailed instructions, conducting contextual analysis, and creating coherent and contextually appropriate original text. For example, previous generations of language models could not write an email from a list of keywords—a task that is trivial for contemporary LLMs. LLMs have remarkable capabilities to understand, generate, and interpret human language. However, it's important to clarify that when we say language models "understand," we mean that they can process and generate text in ways that appear coherent and contextually relevant, not that they possess human-like consciousness or comprehension. Enabled by advancements in deep learning, which is a subset of machine learning and artificial intelligence (AI) focused on neural networks, LLMs are trained on vast quantities of text data. This large-scale training allows LLMs to capture deeper contextual information and subtleties of human language compared to previous approaches. As a result, LLMs have significantly improved performance in a wide range of NLP tasks, including text translation, sentiment analysis, question answering, and many more. The success behind LLMs can be attributed to the transformer architecture that underpins many LLMs and the vast amounts of data on which LLMs are trained, allowing them to capture a wide variety of linguistic nuances, contexts, and patterns that would be challenging to encode manually. This shift toward implementing models based on the transformer architecture and using large training datasets to train LLMs has fundamentally transformed NLP, providing more capable tools for understanding and interacting with human language.

Training large language models (LLMs) requires immense computational power. These models have billions of parameters, and training them involves complex algorithms running on powerful hardware (like GPUs) for days or even months. Cloud services offering this infrastructure come at a significant cost, with factors like compute time, storage space, and data transfer contributing to the overall expense. For example, the compute cost for training GPT-3 alone is estimated to range from about $500,000 to as high as $4.6 million, depending on the specific

hardware and operational efficiencies achieved during the training process. According to a report by Jensen Huang at the NVIDIA GTC 2024, training GPT-MoE-1.8T model using 25,000 Ampere-based GPUs (most likely the A100) took 3 to 5 months. Doing the same with Hopper (H100) would take about 8,000 GPUs in 90 days. Due to the significant financial investment required, most people don't train LLMs from scratch. Instead, they leverage pre-trained models offered by other companies or organizations (like ChatGPT or Llama2).

The large training datasets for popular GPT- and BERT-like models represent diverse and comprehensive text corpora encompassing billions of words, which include a vast array of topics and natural and computer languages. To provide a concrete example, table 1.1 summarizes the dataset used for pretraining GPT-3, which served as the base model for the first version of ChatGPT.

Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

Table 3.1: GPT-3 Training Data

The cost of training depends on various factors:

- Model Architecture
    - Size and Structure

      The depth (number of layers), width (neurons per layer), and the total number of parameters affect both GPU memory requirements and training time. A model with more and/or wider layers has the capacity to learn more complex features, but at the expense of increased computational demand. Increasing the total number of parameters to train increases the estimated time to train and the GPU memory requirements.

    - Attention Mechanisms

      Transformers leverage self-attention mechanisms, with multiple heads attending to different sequence parts, enhancing learning at the cost of increased computation. The traditional Transformer attention style compares every token in the context window with every other token, leading to memory requirements that are quadratic in the size of the context window, $O(n^2)$. Sparse attention models offer a compromise by focusing on a subset of positions, for example with local (nearby) attention, thereby reducing computational load, often down to $O(n\sqrt{n})$

    - Efficiency Optimizations

      Choices of activation functions and gating mechanisms can impact computational intensity and training time. Different activation functions have varying levels of mathematical complexity; ReLU, for example, is less complex than sigmoid or tanh.

- Training Dynamics
    - Learning Rate and Batch Size

      Learning rate and batch size significantly influence the model's training speed and stability. The learning rate of a model affects the step size it takes in the opposite direction of the gradient (i.e. the direction towards minimizing the cost or loss function). This is called gradient descent. The batch size is the number of samples processed before the model's parameters are updated. It is true that the larger your batch, the more memory you need; it scales linearly with the size of the batch. Even if you had a terabyte of GPU memory, you still may not want to use the largest batch size possible. Downsampling (i.e. using a smaller batch size than the total number of training samples) introduces noise into the gradient, which can help you avoid local minima. That's why it's called stochastic gradient descent: the stochasticity refers to how much you're down sampling from your training set in each batch. The learning rate's size (magnitude) and schedule (rate of change over training) can affect the speed and stability of convergence. A higher learning rate means the model takes bigger steps during gradient descent. While this can speed up convergence, it can also lead to overshooting minima and potentially unstable training. Conversely, a learning rate that is too small can slow down convergence (as getting to a minimum takes longer), and the model may get stuck in local minima.

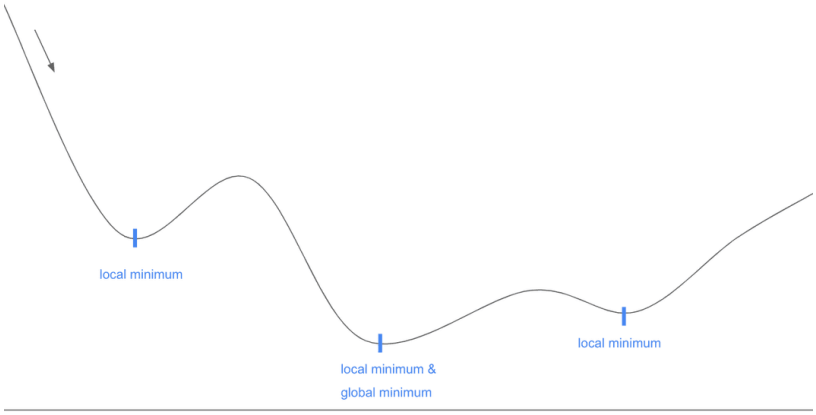


Figure 3.1: A graph showing global minima problem

This figure for an example of local vs. global minima. In simple terms, a local minimum that is not equal to the global minimum is a location on the graph where it seems like the optimal loss has been found, but we had just gone a little further - up a hill and dealing with some worse performance to get there - we could have found a better place in the graph.

- Precision and Quantization

  The precision of calculations, like FP16 versus FP32 - using 16 bits to represent each floating point versus 32 - and techniques such as quantization balance memory usage with performance trade-offs. Using half-precision (FP16) instead of single-precision (FP32) floating points cuts the tensor sizes in half, which can save memory and speed up training by enabling faster operations and more parallelization. However, this comes with a trade-off in precision, which can lead to potential numerical errors, like overflow/underflow errors, as fewer bits can't represent as large or as small numbers. It can also reduce accuracy, but if not too extreme, it can serve as a form of regularization, reducing overfitting and allowing the model to actually perform better on the held-out dataset. Another technique is to use mixed precision training, where some floating points are FP16 and some are FP32. Determining which matrices should be represented as FP16 vs. FP32 may take some experimentation, however, which is also a cost consideration.

  Quantization is another technique that maps high-precision floating points to lower-precision values, usually 8- or even 4-bit fixed-point representation integers. This reduces tensor sizes by 75% or even 87.5%, but usually results in a significant reduction in model accuracy; as mentioned before, though, it may actually help the model generalize better, so experimentation may be worthwhile.

- HyperParameter Sweeps

  Hyperparameters are external configuration variables for machine learning models, i.e. they aren't learned by the model itself, like weights

are. Hyperparameters are basically all the variables we discussed here: learning rate, model architecture like number of neurons or layers, attention mechanisms, etc. Hyperparameter sweeps are when experiments are run training different models with combinations of various hyperparameter settings, and they enable a model to find the best possible combinations of hyperparameter values for its specific dataset and task. However, it is computationally expensive, as you must train many models to find the best configuration.

Pre-Training models requires a lot of computational resources due to the above-mentioned reasons, potentially limiting its accessibility to Organizations and Individuals with limited resources, creating a barrier to entry for smaller players in the field. Therefore, organizations usually opt in for using already pre-trained models either by hosting it on their own infrastructure or using services from IaaS companies on hourly costing for their GPUs or Token-based pricing.

Provider	Model	input price per 1k Token	output price per 1K Token
(Azure) OpenAI	GPT-4 (8K)	$30.00	$60.00
	GPT-4 Turbo	$10.00	$30.00
	GPT-3.5-turbo	$0.50	$1.50
Anthropic	Claude 3 (Opus)	$15.00	$75.00
	Claude 3 (Haiku)	$0.25	$1.25
Google (Vertex AI)	Gemini Pro	$1.00	$2.00
Amazon Bedrock	Claude 3 (Sonnet) (Same as Anthropic)	$3.00	$15.00
	Cohere (Same as on Cohere)	$1.50	$2.00

	Llama 2 70b	$1.95	$2.56
Mistral	Mistral-large	$8.00	$24.00
	Mistral-medium	$2.70	$8.10
	open mistral (7B)	$0.25	$0.25
	open-mixtral-8x7b	$0.70	$0.70
Anyscale	Mistral 7B	$0.15	$0.15
	Mistral-small (8x7B)	$0.50	$0.50
	Llama 2 70b	$1.00	$1.00
MosaicML	Llama 2 70b	$2.00	$2.00
TogetherAI	Mistral-small (8x7B)	$0.60	$0.60
	Llama 2 70b	$0.90	$0.90

Table 3.2: Token Based Pricing for SOTA Models offered by Industry leading Organizations

Research has shown that when it comes to modeling performance, custom-built LLMs—those tailored for specific tasks or domains—can outperform general-purpose LLMs, such as those provided by ChatGPT, which are designed for a wide array of applications. Examples of these include BloombergGPT (specialized for finance) and LLMs tailored for medical question answering (see the Further Reading and References section in appendix B for more details).

Using custom-built LLMs offers several advantages, particularly regarding data privacy. For instance, companies may prefer not to share sensitive data with third-party LLM providers like OpenAI due to confidentiality concerns.

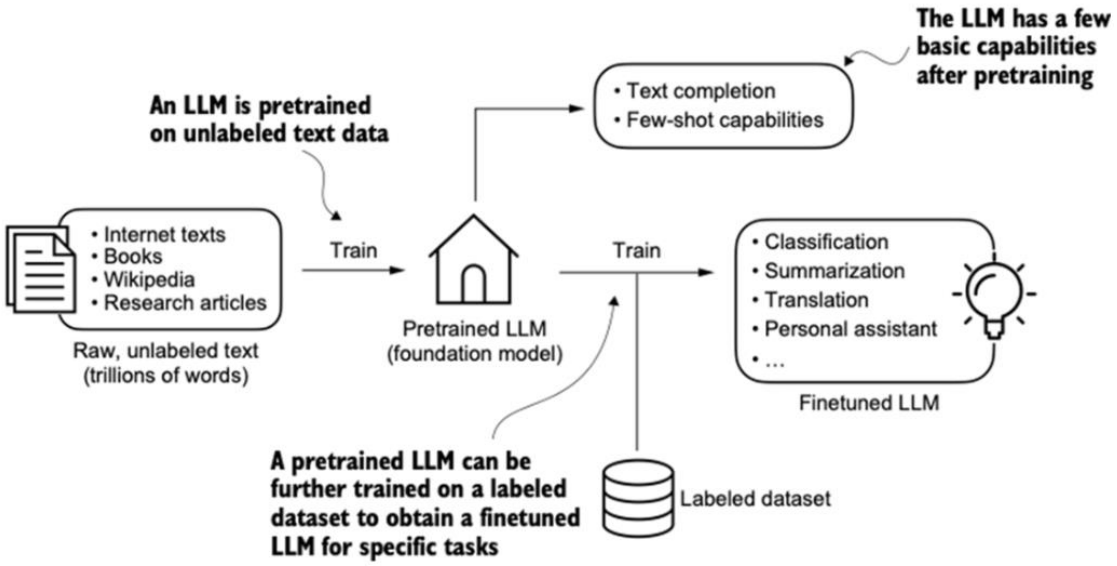


Figure: 3.2: LLM Lifecycle

Pretraining establishes a general-purpose model which has adequate knowledge about almost all domains. But companies/individuals need an expert model for their specific use case, which can be done via leveraging the pretrained knowledge of the LLM with additional learning from a smaller dataset specific to that particular task. This is a cheaper alternative to pre-training a model from scratch.

There are also various challenges faced during fine tuning:

- Data Quality and Quantity

  The success of fine-tuning often depends on the availability and quality of the training data. In some domains, acquiring a sufficient amount of labeled data for specific tasks can be challenging.

- Overfitting and Generalization

  Overfitting, where the model memorizes training data without generalizing well to new data, is a concern. Fine-tuned models might perform exceptionally well on training data but struggle with real-world variations.

- Computational Resources

The vast size and intricate training processes – translate to a high computational burden during fine-tuning. This poses a significant challenge for practitioners with limited resources.

Provider	Model	Cost
AnyScale	Llama 2 7b	$1344
	Llama 2 13b	$2304
	Llama 2 70b	$36480
OpenAI	GPT 3.5 Turbo	$28000

Table 3.3: Cost for training on a Dataset of 3.5M Tokens
OpenAI increases cost for inference of a fine-tuned model

4. METHODOLOGY

The “large” in “large language model” refers to both the model’s size in terms of parameters and the immense dataset on which it’s trained. Models like this often have tens or even hundreds of billions of parameters, which are the adjustable weights in the network that are optimized during training to predict the next word in a sequence. Next-word prediction is sensible because it harnesses the inherent sequential nature of language to train models on understanding context, structure, and relationships within text. Yet, it is a very simple task, and so it is surprising to many researchers that it can produce such capable models.

Since LLMs are capable of generating text, LLMs are also often referred to as a form of generative artificial intelligence (AI), often abbreviated as generative AI or GenAI. AI encompasses the broader field of creating machines that can perform tasks requiring human-like intelligence, including understanding language, recognizing patterns, and making decisions, and includes subfields like machine learning and deep learning.

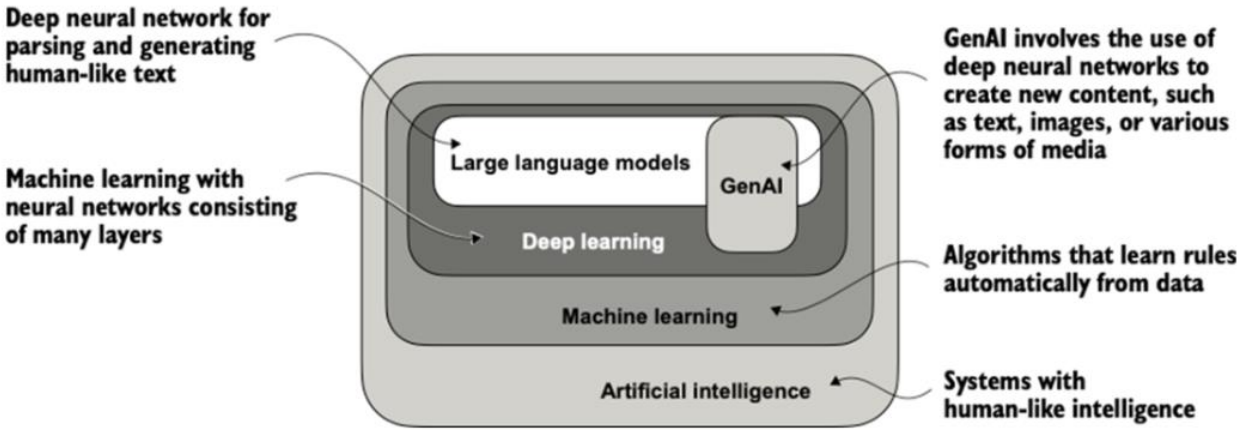


Figure: 4.1: Hierarchical Structure of Artificial Intelligence

The algorithms used to implement AI are the focus of the field of machine learning. Specifically, machine learning involves the development of algorithms that can learn from and make predictions or decisions based on data without being explicitly programmed. To illustrate this, imagine a spam filter as a practical application of machine learning. Instead of manually writing rules to identify spam emails, a machine learning algorithm is fed examples of emails labeled as spam and legitimate emails. By minimizing the error in its predictions on a training dataset, the model

then learns to recognize patterns and characteristics indicative of spam, enabling it to classify new emails as either spam or legitimate.

As illustrated in fig 4.1, deep learning is a subset of machine learning that focuses on utilizing neural networks with three or more layers (also called deep neural networks) to model complex patterns and abstractions in data. In contrast to deep learning, traditional machine learning requires manual feature extraction. This means that human experts need to identify and select the most relevant features for the model.

While the field of AI is now dominated by machine learning and deep learning, it also includes other approaches—for example, using rule-based systems, genetic algorithms, expert systems, fuzzy logic, or symbolic reasoning.

Returning to the spam classification example, in traditional machine learning, human experts might manually extract features from email text such as the frequency of certain trigger words ("prize," "win," "free"), the number of exclamation marks, use of all uppercase words, or the presence of suspicious links. This dataset, created based on these expert-defined features, would then be used to train the model. In contrast to traditional machine learning, deep learning does not require manual feature extraction. This means that human experts do not need to identify and select the most relevant features for a deep learning model. (However, both traditional machine learning and deep learning for spam classification still require the collection of labels, such as spam or non-spam, which need to be gathered either by an expert or users.)

Most modern LLMs rely on the transformer architecture, which is a deep neural network architecture introduced in the 2017 paper "Attention Is All You Need" [**y**]To understand LLMs, we briefly have to go over the original transformer, which was originally developed for machine translation, translating English texts to German and French.

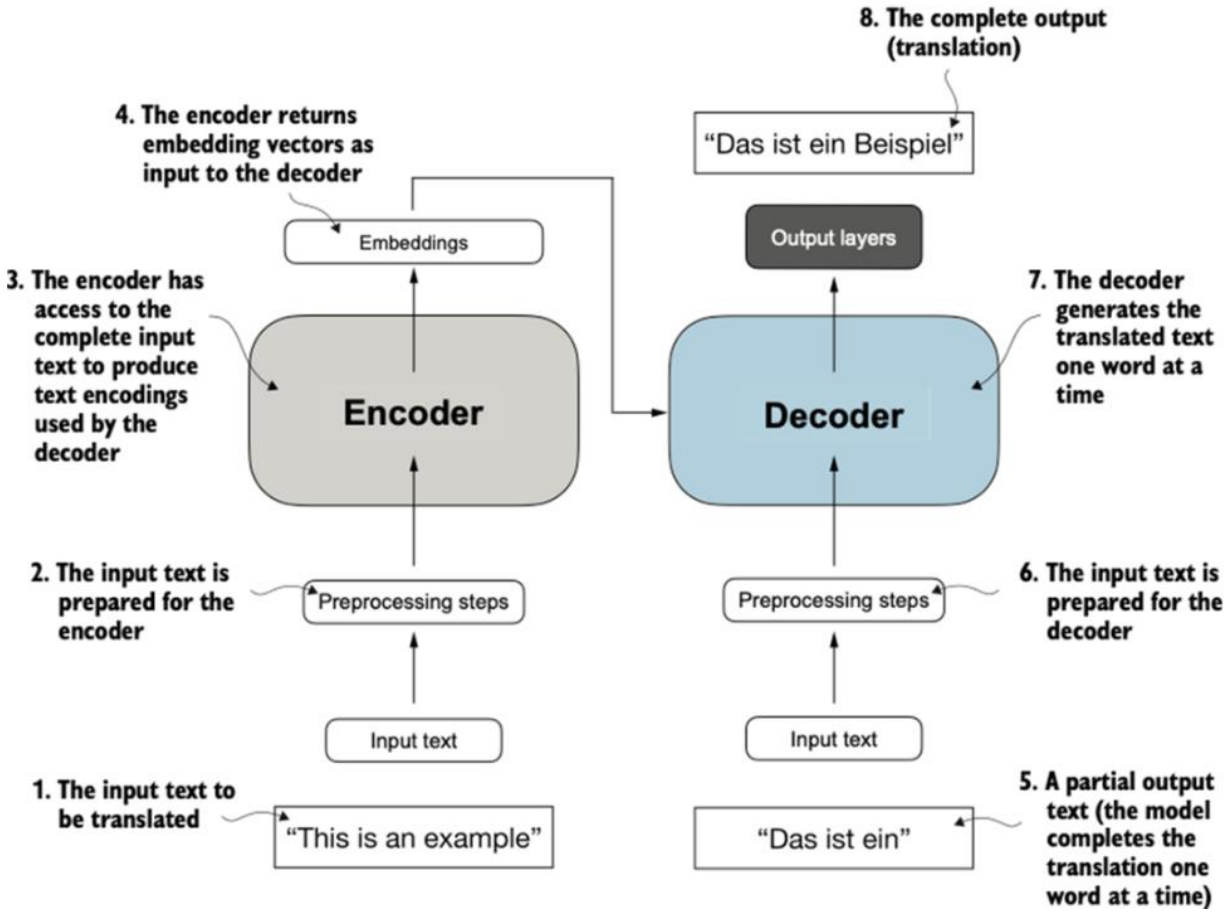


Figure 4.2: Transformer Architecture

The transformer architecture depicted in figure 1.4 consists of two submodules: an encoder and a decoder. The encoder module processes the input text and encodes it into a series of numerical representations or vectors that capture the contextual information of the input. Then, the decoder module takes these encoded vectors and generates the output text. In a translation task, for example, the encoder would encode the text from the source language into vectors, and the decoder would decode these vectors to generate text in the target language. Both the encoder and decoder consist of many layers connected by a so-called self-attention mechanism. You may have many questions regarding how the inputs are preprocessed and encoded.

A key component of transformers and LLMs is the self-attention mechanism (not shown), which allows the model to weigh the importance of different words or tokens in a sequence relative to each other. This mechanism enables the model to capture long-range dependencies and contextual relationships within the input data, enhancing its ability to generate coherent and contextually relevant output.

Later variants of the transformer architecture, such as the so-called BERT (short for bidirectional encoder representations from transformers) and the various GPT models (short for generative pretrained transformers), built on this concept to adapt this architecture for different tasks.

BERT, which is built upon the original transformer's encoder submodule, differs in its training approach from GPT. While GPT is designed for generative tasks, BERT and its variants specialize in masked word prediction, where the model predicts masked or hidden words in a given sentence, as illustrated in figure 4.3. This unique training strategy equips BERT with strengths in text classification tasks, including sentiment prediction and document categorization. As an application of its capabilities, as of this writing, X (formerly Twitter) uses BERT to detect toxic content.

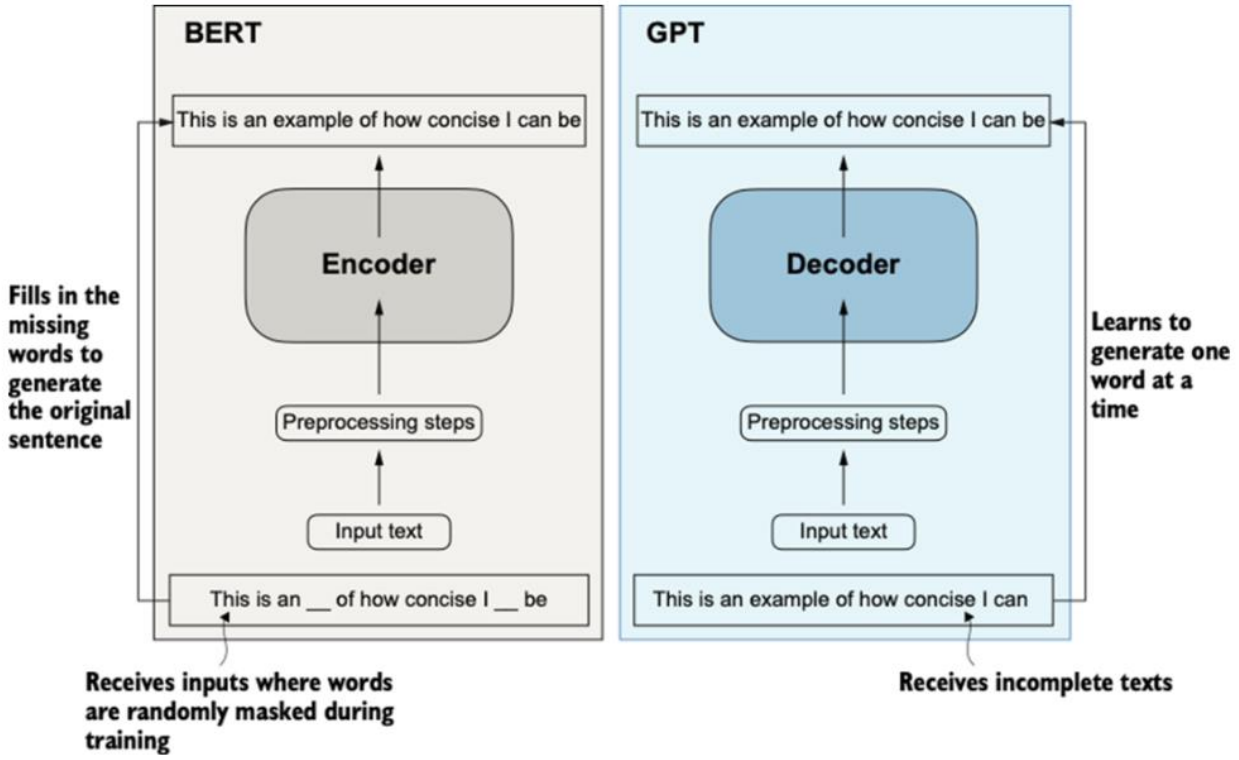


Figure 4.3: BERT vs GPT Architecture

GPT, on the other hand, focuses on the decoder portion of the original transformer architecture and is designed for tasks that require generating texts. This includes machine translation, text summarization, fiction writing, writing computer code, and more. We will discuss the GPT architecture in more detail in the remaining sections of this chapter and implement it from scratch in this book.

GPT models, primarily designed and trained to perform text completion tasks, also show remarkable versatility in their capabilities. These models are adept at executing both zero-shot and few-shot learning tasks. Zero-shot learning refers to the ability to generalize to completely unseen tasks without any prior specific examples. On the other hand, few-shot learning involves learning from a minimal number of examples the user provides as input,

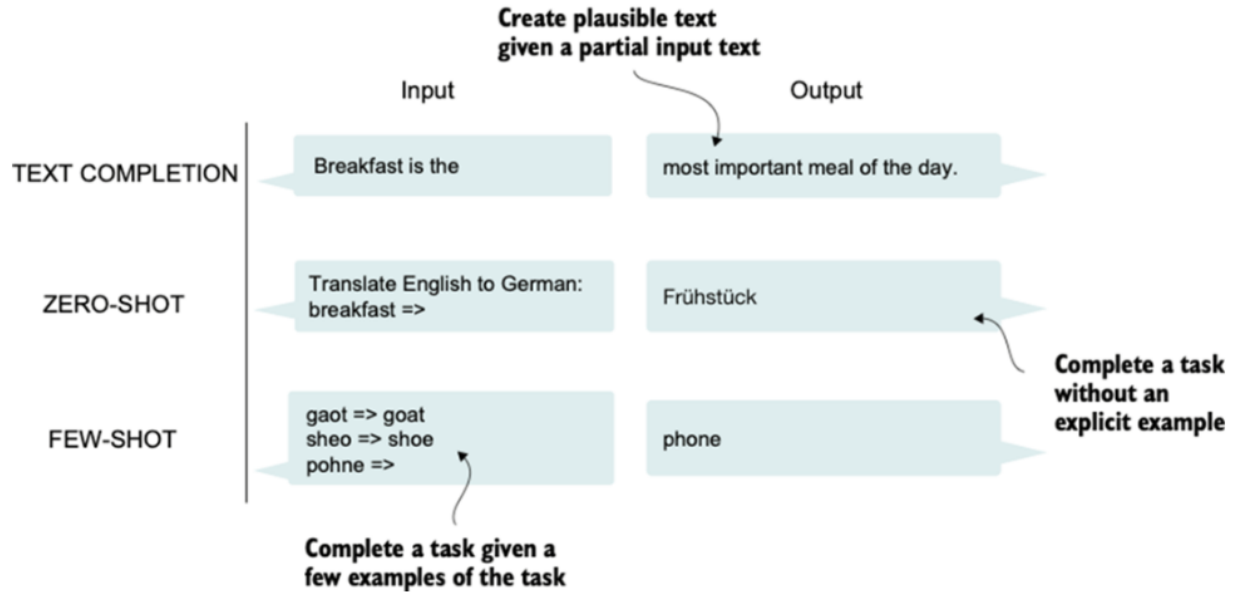


Figure 4.4: Example of Text completions

Since revolution of LLMs emerged from GPT , we will have a closer look at the GPT architecture. GPT stands for Generative Pretrained Transformer and was originally introduced in the paper "Improving Language Understanding by Generative Pre-Training" by Radford et al. from OpenAI[**z**]. GPT-3 is a scaled-up version of this model that has more parameters and was trained on a larger dataset. In addition, the original model offered in ChatGPT was created by finetuning GPT-3 on a large instruction dataset using a method from OpenAI's InstructGPT paper, these models are competent text completion models and can carry out other tasks such as spelling correction, classification, or language translation. This is actually very remarkable given that GPT models are pretrained on a relatively simple next-word prediction task. The next-word prediction task is a form of self-supervised learning, which is a form of self-labeling. This means that we don't need to collect labels for the training data explicitly but can leverage the structure of the data itself: we can use the next word in a sentence or document as the label that the model is

supposed to predict. Since this next-word prediction task allows us to create labels "on the fly," it is possible to leverage massive unlabeled text datasets to train LLMs.

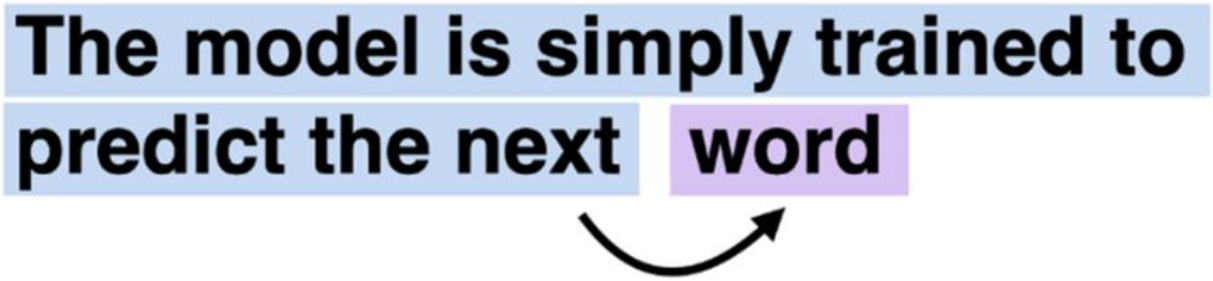


Figure 4.5: Next word prediction

Compared to the original transformer architecture we covered in section 1.4, the general GPT architecture is relatively simple. Essentially, it's just the decoder part without the encoder as illustrated in figure 1.8. Since decoder-style models like GPT generate text by predicting text one word at a time, they are considered a type of autoregressive model. Autoregressive models incorporate their previous outputs as inputs for future predictions. Consequently, in GPT, each new word is chosen based on the sequence that precedes it, which improves the coherence of the resulting text.

Architectures such as GPT-3 are also significantly larger than the original transformer model. For instance, the original transformer repeated the encoder and decoder blocks six times. GPT-3 has 96 transformer layers and 175 billion parameters in total.

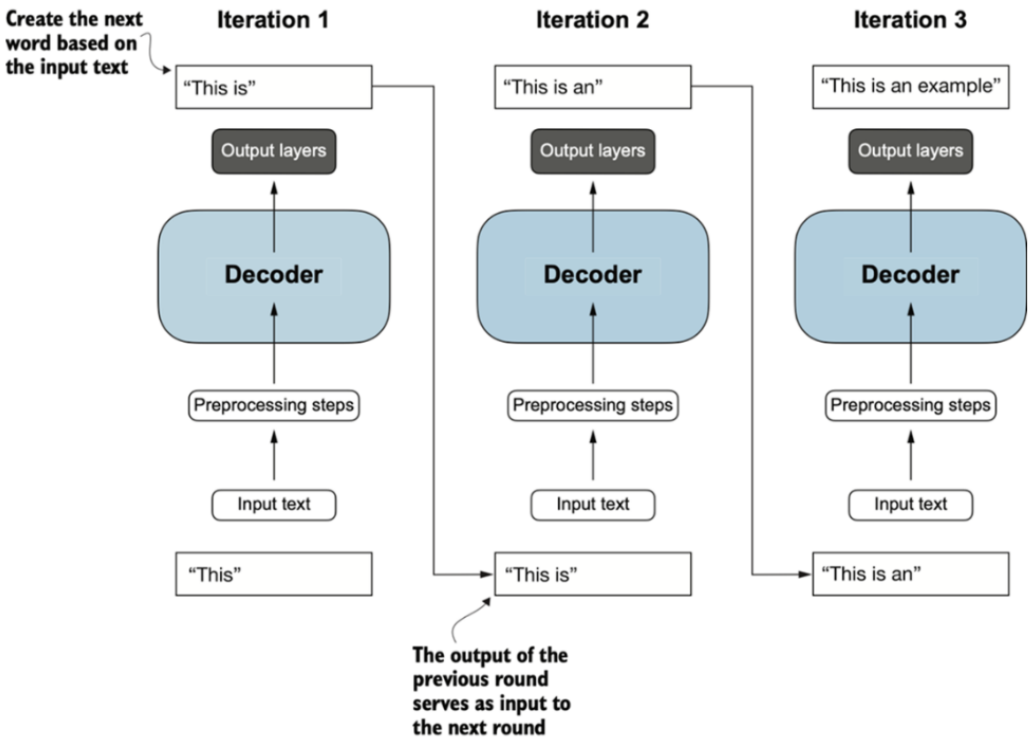


Figure 4.6: Iterations of next word prediction

GPT-3 was introduced in 2020, which, by the standards of deep learning and large language model development, is considered a long time ago. However, more recent architectures, such as Meta's Llama models, are still based on the same underlying concepts, introducing only minor modifications. Hence, understanding GPT remains as relevant as ever, and this book focuses on implementing the prominent architecture behind GPT while providing pointers to specific tweaks employed by alternative LLMs.

Lastly, it's interesting to note that although the original transformer model, consisting of encoder and decoder blocks, was explicitly designed for language translation, GPT models—despite their larger yet simpler decoder-only architecture aimed at next-word prediction—are also capable of performing translation tasks. This capability was initially unexpected to researchers, as it emerged from a model primarily trained on a next-word prediction task, which is a task that did not specifically target translation.

The ability to perform tasks that the model wasn't explicitly trained to perform is called an emergent behavior. This capability isn't explicitly taught during training but emerges as a natural consequence of the model's exposure to vast quantities of

multilingual data in diverse contexts. The fact that GPT models can "learn" the translation patterns between languages and perform translation tasks even though they weren't specifically trained for it demonstrates the benefits and capabilities of these large-scale, generative language models. We can perform diverse tasks without using diverse models for each.

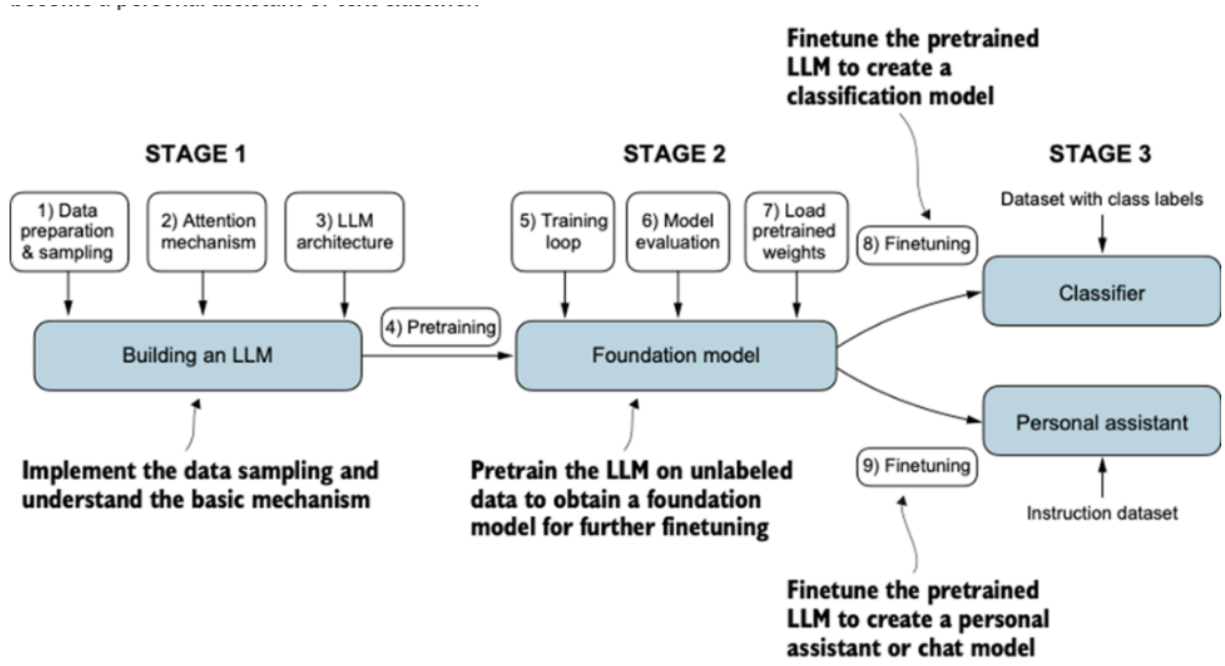


Figure 4.7: Steps to build a LLM

During the pretraining stage, LLMs process text one word at a time. Training LLMs with millions to billions of parameters using a next-word prediction task yields models with impressive capabilities. These models can then be further finetuned to follow general instructions or perform specific target tasks. But before we can implement and train LLMs in the upcoming chapters, we need to prepare the training dataset.

Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with the mathematical operations used to implement and train neural networks. Therefore, we need a way to represent words as continuous-valued vectors. The concept of converting data into a vector format is often referred to as embedding. Using a specific neural network layer or another pretrained neural network model, we can embed different data types—for example, video, audio, and text.

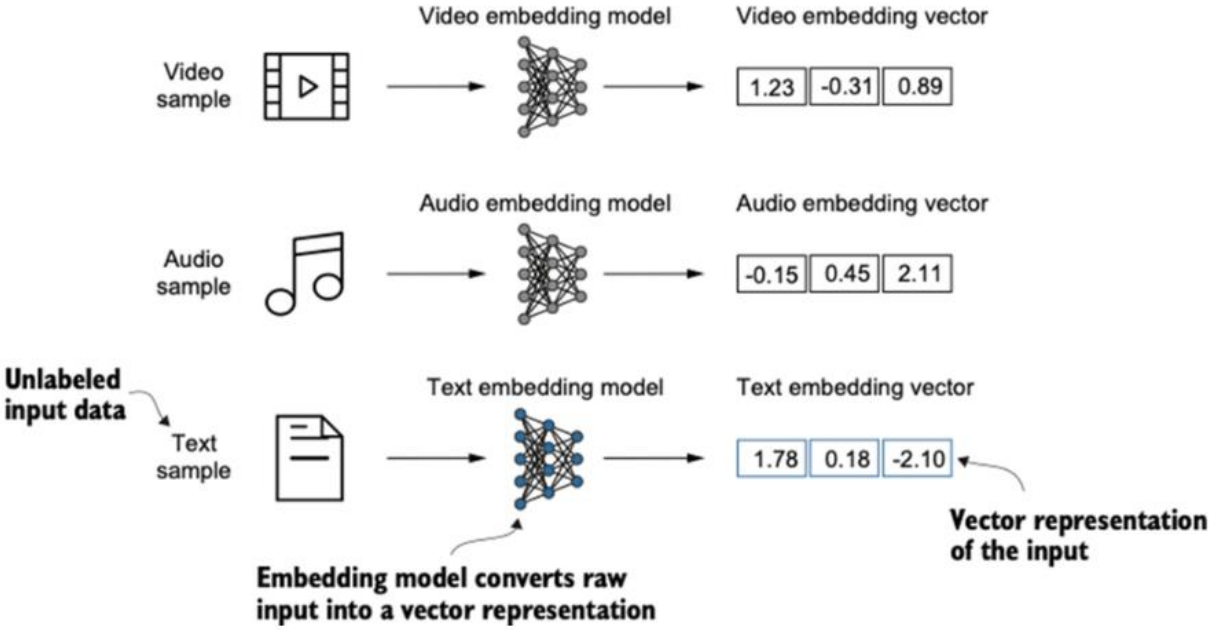


Figure 4.8: Working of Embeddings on different data

However, it's important to note that different data formats require distinct embedding models. For example, an embedding model designed for text would not be suitable for embedding audio or video data. At its core, an embedding is a mapping from discrete objects, such as words, images, or even entire documents, to points in a continuous vector space—the primary purpose of embeddings is to convert nonnumeric data into a format that neural networks can process. While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for retrieval-augmented generation. Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generating text, which is a technique that is beyond the scope of this study. Several algorithms and frameworks have been developed to generate word embeddings. One of the earlier and most popular examples is the Word2Vec approach. Word2Vec trained neural network architecture to generate word embeddings by predicting the context of a word given the target word or vice versa. The main idea behind Word2Vec is that words that appear in similar contexts tend to have similar meanings. Consequently, when projected into two-dimensional word embeddings for visualization purposes, it can be seen that similar terms cluster together.

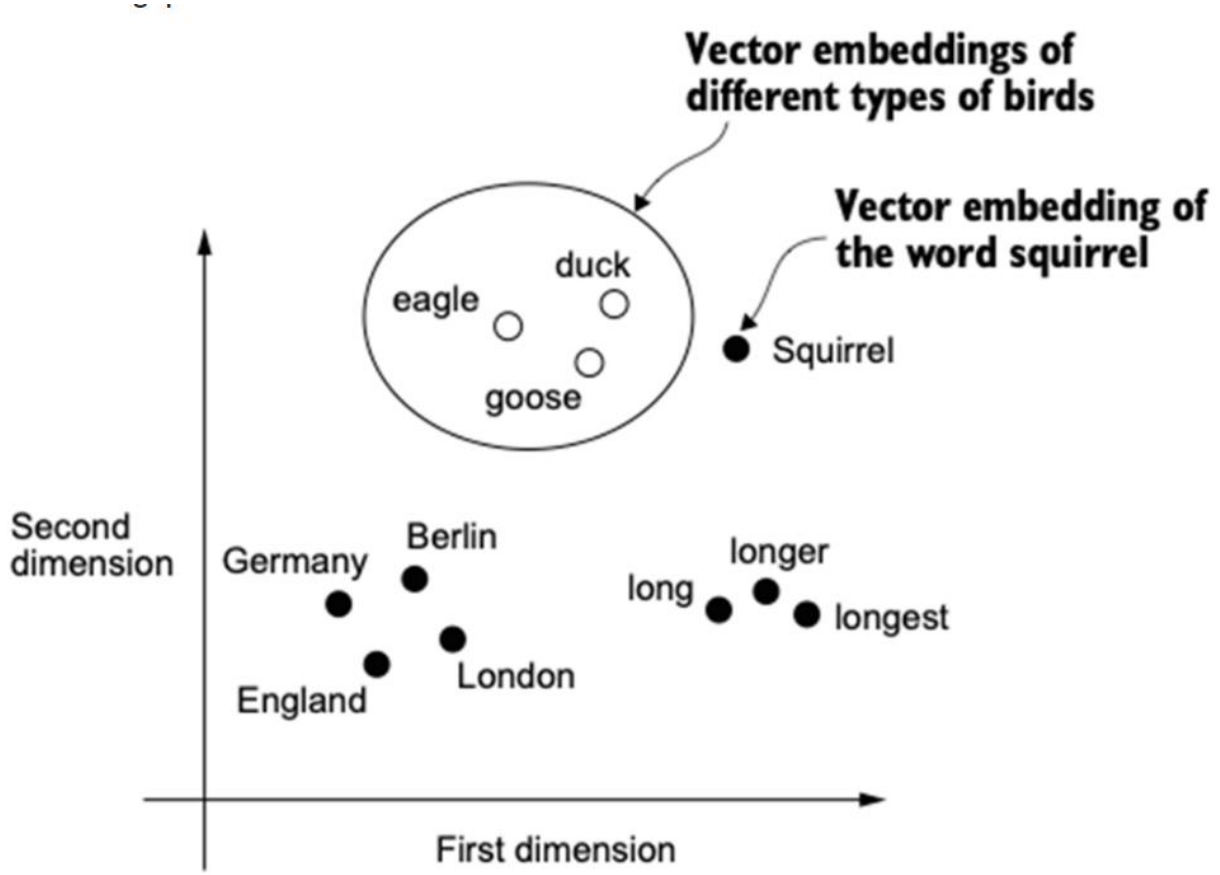


Figure 4.9: Vector Embeddings of words

A higher dimensionality might capture more nuanced relationships but at the cost of computational efficiency.

While we can use pretrained models such as Word2Vec to generate embeddings for machine learning models, LLMs commonly produce their own embeddings that are part of the input layer and are updated during training. The advantage of optimizing the embeddings as part of the LLM training instead of using Word2Vec is that the embeddings are optimized to the specific task and data at hand. Furthermore, LLMs can also create contextualized output embeddings.

For both GPT-2 and GPT-3, the embedding size (often referred to as the dimensionality of the model's hidden states) varies based on the specific model variant and size. It is a trade-off between performance and efficiency. The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

Language models process text in discrete units called tokens. Tokens can represent words, subwords, or even characters, depending on the tokenization method used. Tokenization provides a structured way to break down text into manageable pieces for the model to process. we know that large language models are pre-trained on a huge corpus of text (think of all Wikipedia pages). Tokenization takes the corpus and provides a vocabulary of tokens that exist in the corpus. The simplest way is to split the corpus by whitespace and output words as vocabulary. That would give a very large vocabulary. For example, the number of words in the English Wikipedia corpus must be around 2 billion. Tokenization helps manage the vocabulary by mapping text elements to predefined tokens in the model's vocabulary. This is essential for controlling memory and computational requirements.

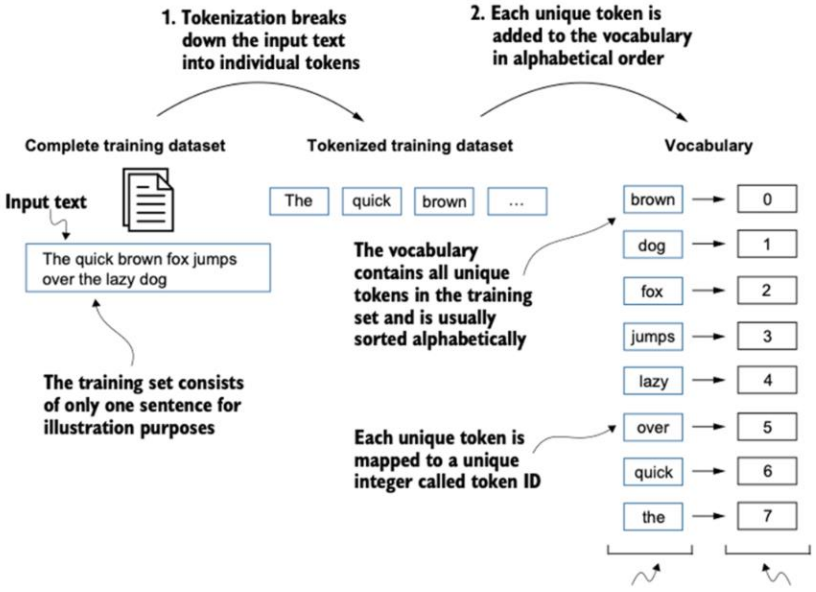


Figure 4.10: Vocabulary creation by Tokenzation

It is the first step and the last step of text processing and modeling. Texts need to be represented as numbers in our models so that our model can understand. Tokenization breaks down text into tokens, and each token is assigned a numerical representation, or index, which can be used to feed into a model. The LLM first encode the input text into tokens using a tokenizer. Each unique token is assigned a specific index number in the tokenizer's vocabulary.

the text is tokenized, these tokens are passed through the model, which typically includes an embedding layer and transformer blocks. The embedding layer converts the tokens into dense vectors that capture semantic meanings. The last step is decoding, which detokenize output tokens back to human-readable text. This is done

by mapping the tokens back to their corresponding words using the tokenizer's vocabulary.

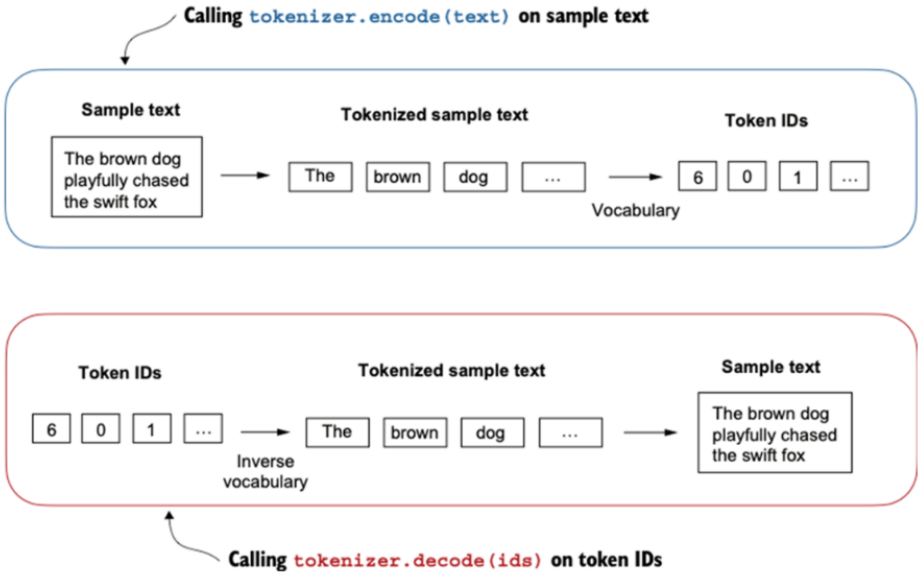


Figure 4.11: Tokenization Cycle

The problem is that the word "Hello" was not used in the "The Verdict" short story. Hence, it is not contained in the vocabulary. This highlights the need to consider large and diverse training sets to extend the vocabulary when working on LLMs.

We will also discuss the usage and addition of special context tokens that can enhance a model's understanding of context or other relevant information in the text. These special tokens can include markers for unknown words and document boundaries, for example. We can modify the vocabulary and tokenizer we implemented in the previous section, SimpleTokenizerV2, to support two new tokens, <|unk|> and <|endoftext|>

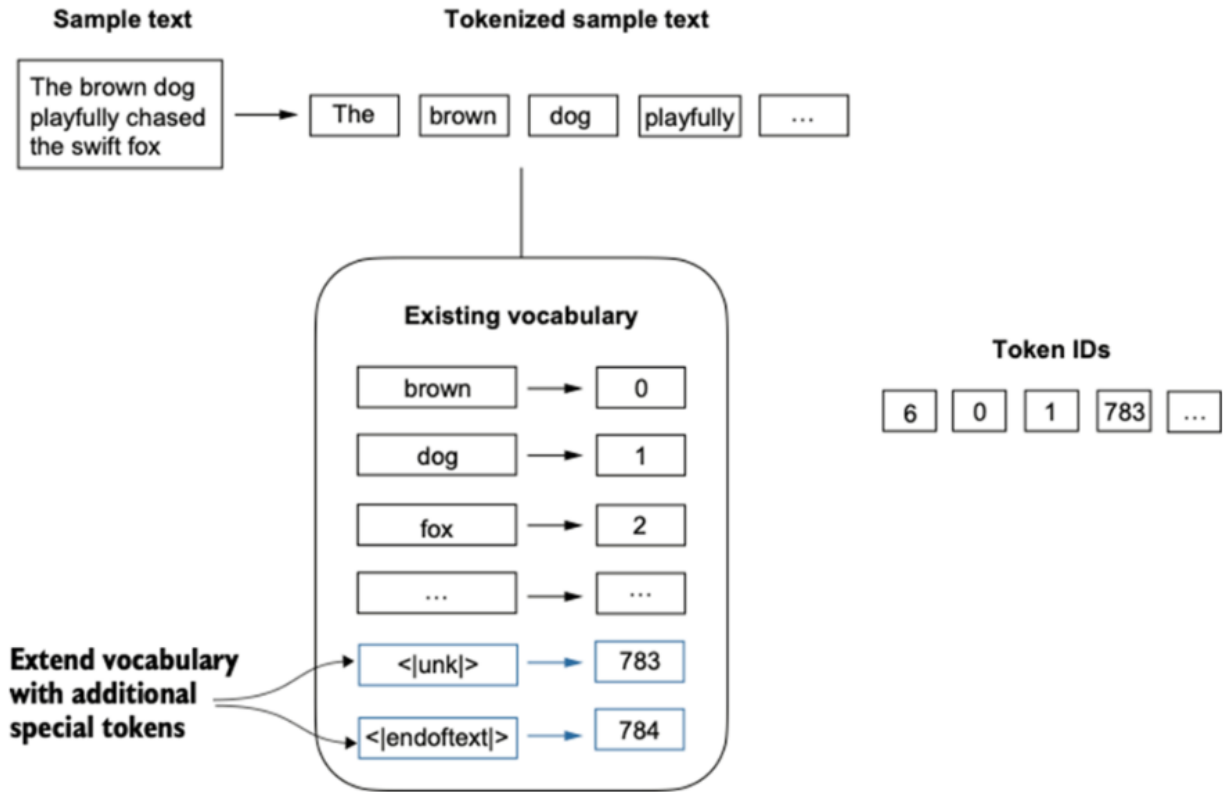


Figure 4.12: Token ID Creation

we can modify the tokenizer to use an <|unk|> token if it encounters a word that is not part of the vocabulary. Furthermore, we add a token between unrelated texts. For example, when training GPT-like LLMs on multiple independent documents or books, it is common to insert a token before each document or book that follows a previous text source. This helps the LLM understand that, although these text sources are concatenated for training, they are, in fact, unrelated.

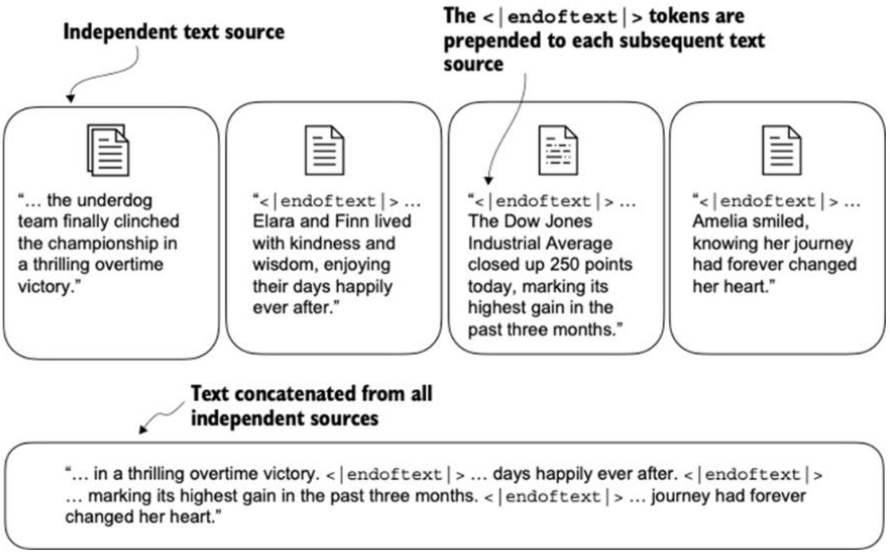


Figure 4.13: Special Tokens

Depending on the LLM, some researchers also consider additional special tokens such as the following:

[BOS] (beginning of sequence)—This token marks the start of a text. It signifies to the LLM where a piece of content begins.

[EOS] (end of sequence)—This token is positioned at the end of a text and is especially useful when concatenating multiple unrelated texts, similar to <|endoftext|>. For instance, when combining two different Wikipedia articles or books, the [EOS] token indicates where one article ends and the next one begins.

[PAD] (padding)—When training LLMs with batch sizes larger than one, the batch might contain texts of varying lengths. To ensure all texts have the same length, the shorter texts are extended or "padded" using the [PAD] token, up to the length of the longest text in the batch.

Note that the tokenizer used for GPT models does not need any of these tokens mentioned here but only uses an <|endoftext|> token for simplicity. The <|endoftext|> is analogous to the [EOS] token mentioned earlier. Also, <|endoftext|> is used for padding as well. However, when training on batched inputs, we typically use a mask, meaning we don't attend to padded tokens. Thus, the specific token chosen for padding becomes inconsequential.

Moreover, the tokenizer used for GPT models also doesn't use an <|unk|> token for out-of-vocabulary words. Instead, GPT models use a byte pair encoding tokenizer, which breaks words down into subword units

The algorithm underlying BPE breaks down words that aren't in its predefined vocabulary into smaller subword units or even individual characters, enabling it to handle out-of-vocabulary words. So, thanks to the BPE algorithm, if the tokenizer encounters an unfamiliar word during tokenization, it can represent it as a sequence of subword tokens or characters.

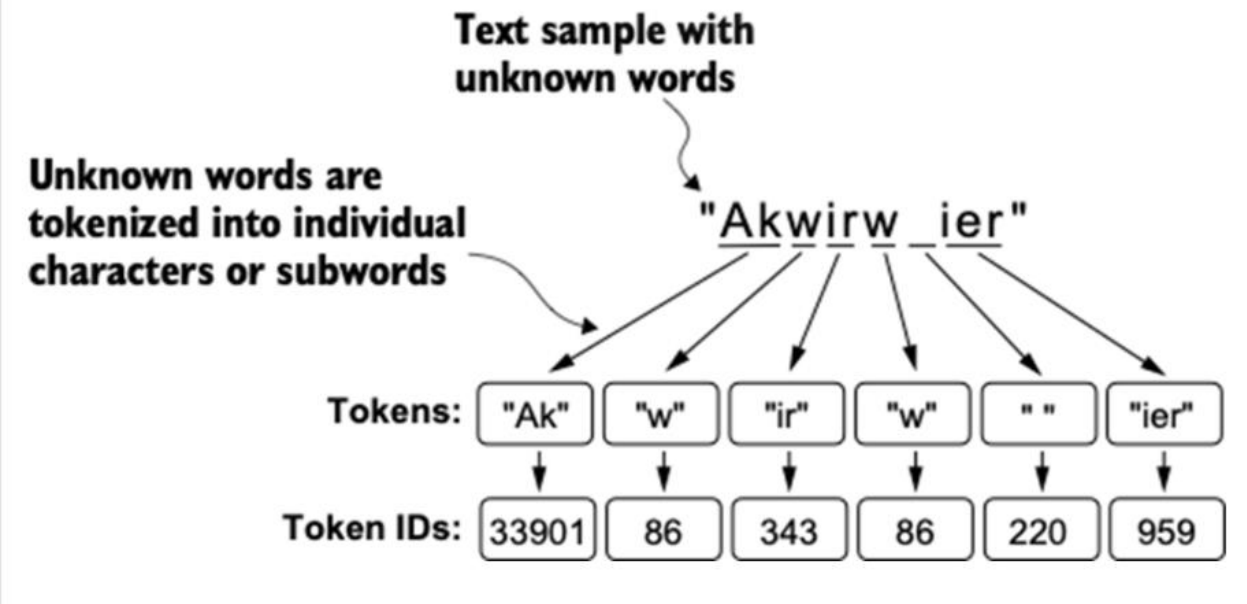


Figure 4.14: Byte Pair Encoding

The ability to break down unknown words into individual characters ensures that the tokenizer, and consequently the LLM that is trained with it, can process any text, even if it contains words that were not present in its training data.

Token learner algorithm

- Initialization:
  - Vocabulary $\mathcal{V} \leftarrow$ unique characters in text.
  - Tokenize text into separate characters.
- for iteration $i = 1$ to K:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

- iteration $i = 1$:
  - Find most frequent pair of adjacent tokens: t_L, t_R
  - Merge tokens: $t_{\text{new}} = t_L t_R$
  - Add to vocabulary: $\mathcal{V} \leftarrow \mathcal{V} \cup \{t_{\text{new}}\}$
  - Replace all occurrences of t_L, t_R in text with t_{new}.

Figure 4.15: Token Learning Algorithm

The next step is to generate the input-target pairs required for an LLM training. LLMs are pretrained by predicting the next word in a text as shown in fig 4.14

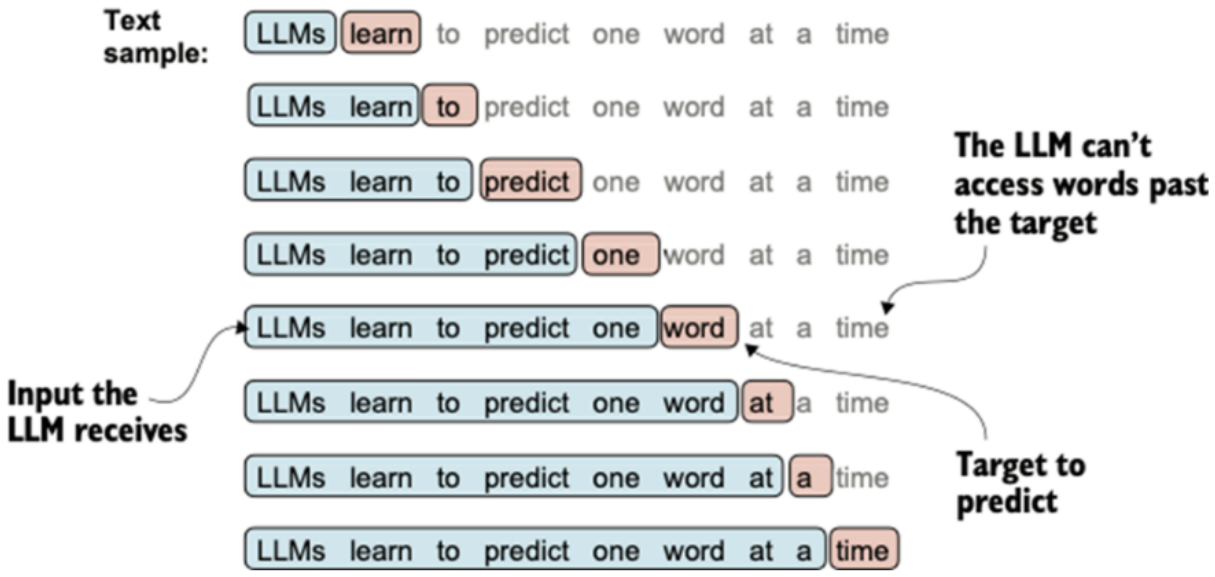


Figure 4.16: Input-Target text pair

To get the pairs we implement a dataloader that fetches them using a sliding window approach. One of the easiest and most intuitive ways to create the input-target pairs for the next-word prediction task is to create two variables, x and y, where x contains the input tokens and y contains the targets, which are the inputs shifted by 1. Processing the inputs along with the targets, which are the inputs shifted by one position, we can then create the next-word prediction tasks.

Everything left of the arrow (---->) refers to the input an LLM would receive, and the token ID on the right side of the arrow represents the target token ID that the LLM is supposed to predict.

and ----> established

and established ----> himself

and established himself ----> in

and established himself in ----> a

There's only one more task before we can turn the tokens into embeddings, that is implementing an efficient data loader that iterates over the input dataset and returns the inputs and targets as PyTorch tensors, which can be thought of as multidimensional arrays. In particular, we are interested in returning two tensors: an input tensor containing the text that the LLM sees and a target tensor that includes the targets for the LLM to predict, as depicted in figure 4.17. While figure 4.17 shows the tokens in string format for illustration purposes, the code implementation will operate on token IDs directly since the encode method of the BPE tokenizer performs both tokenization and conversion into token IDs as a single step.

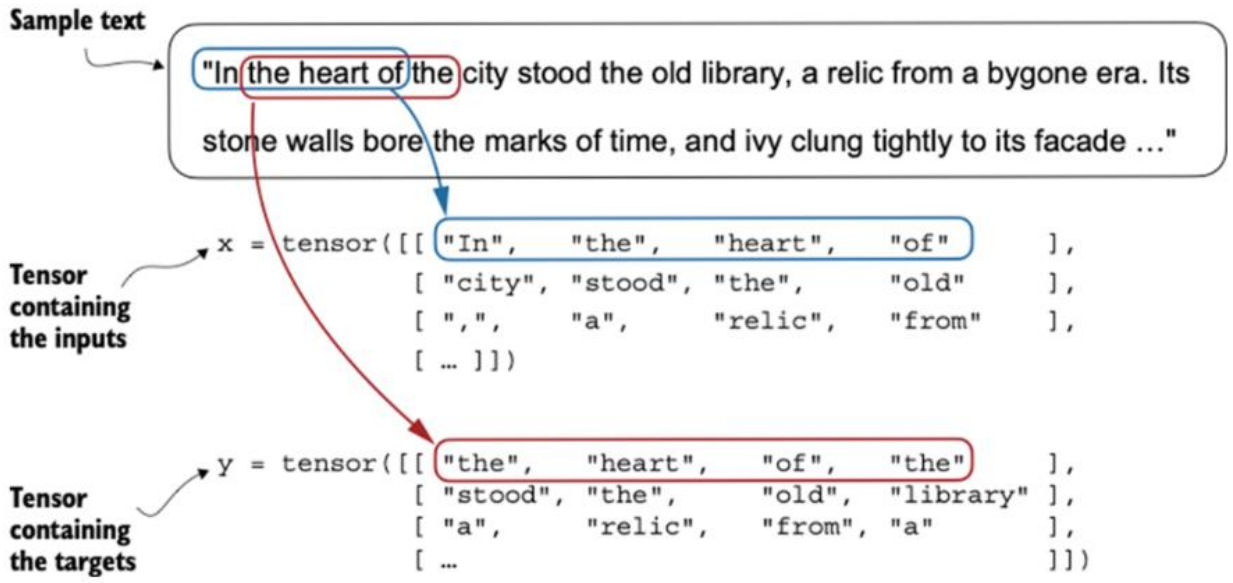


Figure 4.17: Input-Target Tensors

The next step is to convert the token IDs into embedding vectors. it is important to note that we initialize these embedding weights with random values as a preliminary step. This initialization serves as the starting point for the LLM's learning process. We will optimize the embedding weights as part of the LLM training. A continuous

vector representation, or embedding, is necessary since GPT-like LLMs are deep neural networks trained with the backpropagation algorithm.

The embedding layer is essentially a lookup operation that retrieves rows from the embedding layer's weight matrix via a token ID. In principle, this is a suitable input for an LLM. However, a minor shortcoming of LLMs is that their self-attention mechanism. The way the previously introduced embedding layer works is that the same token ID always gets mapped to the same vector representation, regardless of where the token ID is positioned in the input sequence.

In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes. However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

To achieve this, we can use two broad categories of position-aware embeddings: relative positional embeddings and absolute positional embeddings. Absolute positional embeddings are directly associated with specific positions in a sequence. For each position in the input sequence, a unique embedding is added to the token's embedding to convey its exact location. For instance, the first token will have a specific positional embedding, the second token another distinct embedding, and so on.

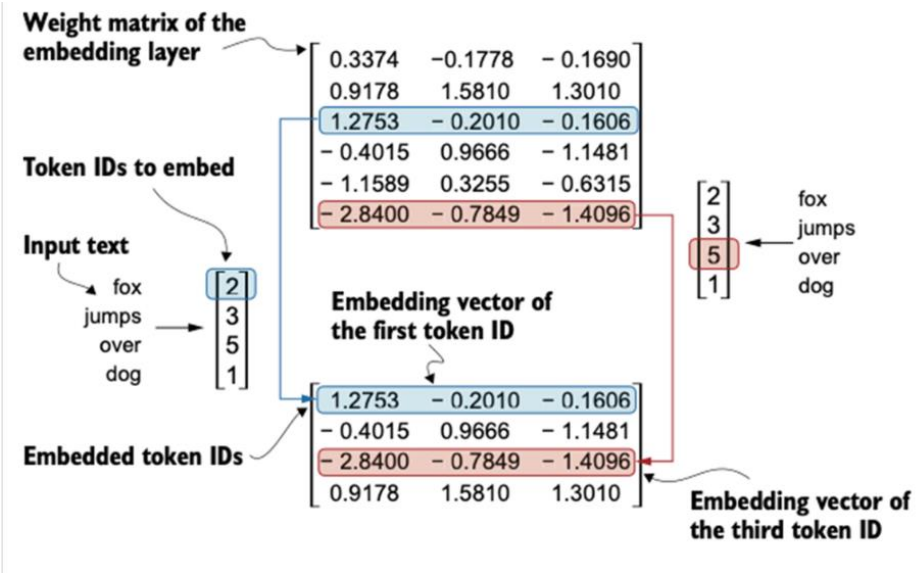


Figure 4.18: Embedding Creation

Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of "how far apart" rather than "at which exact position." The advantage here is that the model can generalize better to sequences of varying lengths, even if it hasn't seen such lengths during training.

Both types of positional embeddings aim to augment the capacity of LLMs to understand the order and relationships between tokens, ensuring more accurate and context-aware predictions. The choice between them often depends on the specific application and the nature of the data being processed.

OpenAI's GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original Transformer model.

Attention mechanisms are a comprehensive topic, which is why we are devoting a whole chapter to it. We will largely look at these attention mechanisms in isolation and focus on them at a mechanistic level. In the next chapter, we will then code the remaining parts of the LLM surrounding the self-attention mechanism to see it in action and to create a model to generate text.

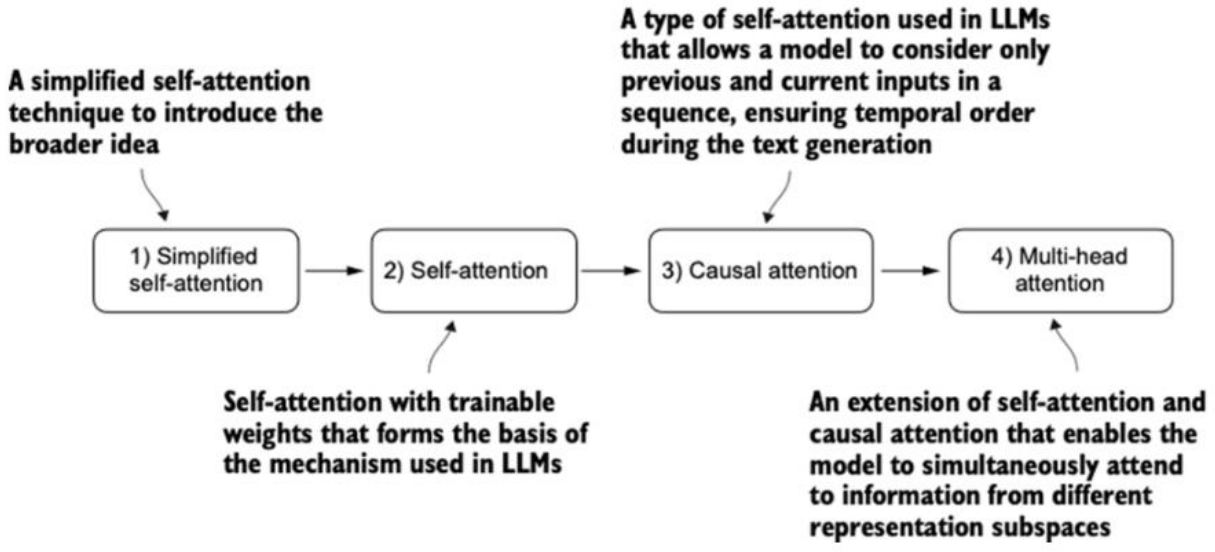


Figure 4.19: Types of Attention

In an encoder-decoder RNN, the input text is fed into the encoder, which processes it sequentially. The encoder updates its hidden state (the internal values at the hidden

layers) at each step, trying to capture the entire meaning of the input sentence in the final hidden state. The decoder then takes this final hidden state to start generating the translated sentence, one word at a time. It also updates its hidden state at each step, which is supposed to carry the context necessary for the next-word prediction.

While we don't need to know the inner workings of these encoder-decoder RNNs, the key idea here is that the encoder part processes the entire input text into a hidden state (memory cell). The decoder then takes in this hidden state to produce the output.

The big limitation of encoder-decoder RNNs is that the RNN can't directly access earlier hidden states from the encoder during the decoding phase. Consequently, it relies solely on the current hidden state, which encapsulates all relevant information. This can lead to a loss of context, especially in complex sentences where dependencies might span long distances.

Before transformer LLMs, it was common to use RNNs for language modeling tasks such as language translation, as mentioned previously. RNNs work fine for translating short sentences but don't work well for longer texts as they don't have direct access to previous words in the input.

One major shortcoming in this approach is that the RNN must remember the entire encoded input in a single hidden state before passing it to the decoder, as illustrated in figure 3.4 in the previous section.

Hence, researchers developed the so-called Bahdanau attention mechanism for RNNs in 2014 (named after the first author of the respective paper), which modifies the encoder-decoder RNN such that the decoder can selectively access different parts of the input sequence at each decoding step

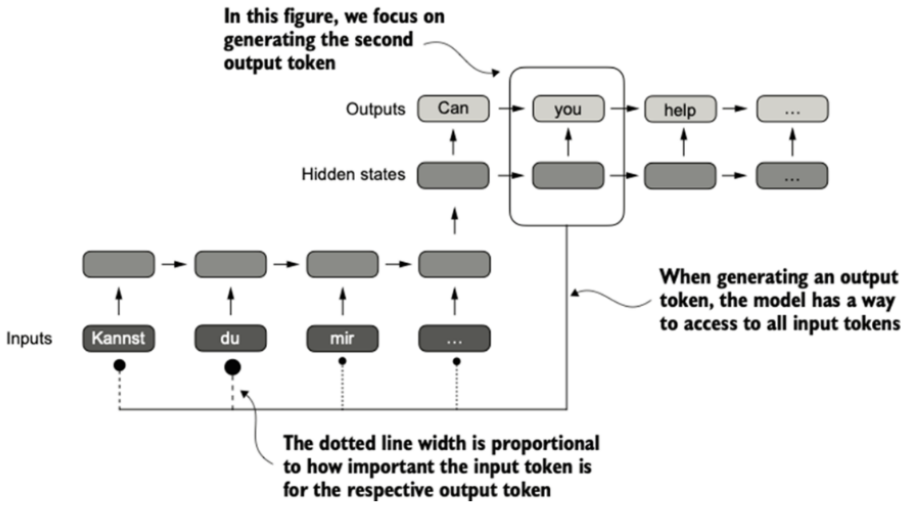


Figure 4.20: Text generating part of the decoder

Interestingly, only three years later, researchers found that RNN architectures are not required for building deep neural networks for natural language processing and proposed the original transformer architecture with a self-attention mechanism inspired by the Bahdanau attention mechanism.

Self-attention is a mechanism that allows each position in the input sequence to attend to all positions in the same sequence when computing the representation of a sequence. Self-attention is a key component of contemporary LLMs based on the transformer architecture, such as the GPT series.

We'll now cover the inner workings of the self-attention mechanism and learn how to code it from the ground up. Self-attention serves as the cornerstone of every LLM based on the transformer architecture.

In this section, we implement a simplified variant of self-attention, free from any trainable weights, summarized in figure 3.7. The goal of this section is to paragraph a few key concepts in self-attention before adding trainable weights

Figure 3.7 shows an input sequence, denoted as x, consisting of T elements represented as x(1) to x(T). This sequence typically represents text, such as a sentence, that has already been transformed into token embeddings

For example, consider an input text like "Your journey starts with one step." In this case, each element of the sequence, such as x(1), corresponds to a d-dimensional

embedding vector representing a specific token, like “Your.” In figure 3.7, these input vectors are shown as three-dimensional embeddings.

In self-attention, our goal is to calculate context vectors z(i) for each element x(i) in the input sequence. A context vector can be interpreted as an enriched embedding vector.

To illustrate this concept, let’s focus on the embedding vector of the second input element, x(2) (which corresponds to the token “journey”), and the corresponding context vector, z(2), shown at the bottom of figure 3.7. This enhanced context vector, z(2), is an embedding that contains information about x(2) and all other input elements x(1) to x(T).

In self-attention, context vectors play a crucial role. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence, as illustrated in figure 3.7. This is essential in LLMs, which need to understand the relationship and relevance of words in a sentence to each other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token. In this section, we implement a simplified self-attention mechanism to compute these weights and the resulting context vector one step at a time.

Consider the following input sentence, which has already been embedded into three-dimensional vectors, as discussed in chapter 2. We choose a small embedding dimension for illustration purposes to ensure it fits on the page without line breaks:

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
```

Figure 4.21: Creating Tensor using PyTorch

The first step of implementing self-attention is to compute the intermediate values ω, referred to as attention scores, as illustrated in figure 3.8. (Please note that figure 3.8 displays the values of the preceding inputs tensor in a truncated version; for example, 0.87 is truncated to 0.8 due to spatial constraints. In this truncated

version, the embeddings of the words “journey” and “starts” may appear similar by random chance.)

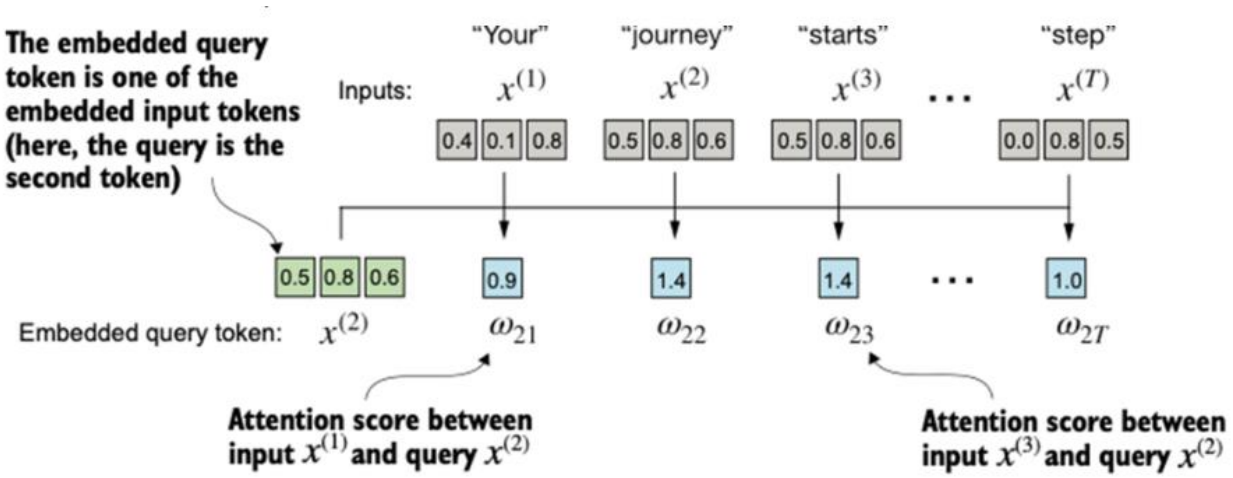


Figure 4.22: Calculation of Attention Score

Figure 3.8 illustrates how we calculate the intermediate attention scores between the query token and each input token. We determine these scores by computing the dot product of the query, x(2), with every other input token:

```
query = inputs[1]
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
```

The computed attention scores are as follows:

```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

Figure 4.23: Code snippet for attention Scores

In the next step, as shown in figure 3.9, we normalize each of the attention scores that we computed previously.

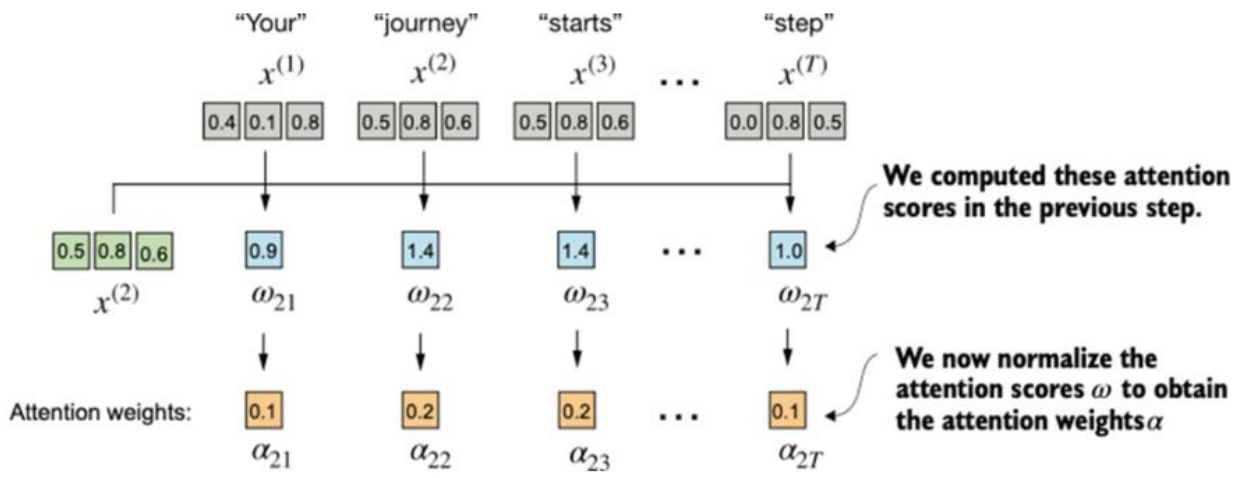


Figure 4.24: Normalizing Attention Scores

The main goal behind the normalization shown in figure 3.9 is to obtain attention weights that sum up to 1. This normalization is a convention that is useful for interpretation and for maintaining training stability in an LLM. Here's a straightforward method for achieving this normalization step:

```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
```

As the output shows, the attention weights now sum to 1:

```
Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
Sum: tensor(1.0000)
```

Figure 4.25: Code for Attention Weights

it's more common and advisable to use the softmax function for normalization. This approach is better at managing extreme values and offers more favorable gradient properties during training. The following is a basic implementation of the softmax function for normalizing the attention scores:

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)

attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
```

As the output shows, the softmax function also meets the objective and normalizes the attention weights such that they sum to 1:

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
Sum: tensor(1.)
```

Figure 4.26: Code for softmax function

In addition, the softmax function ensures that the attention weights are always positive. This makes the output interpretable as probabilities or relative importance, where higher weights indicate greater importance.

Note that this naive softmax implementation (softmax_naive) may encounter numerical instability problems, such as overflow and underflow, when dealing with large or small input values. Therefore, in practice, it's advisable to use the PyTorch implementation of softmax, which has been extensively optimized for performance:

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

Figure 4.27: Implementing softmax with Pytorch

In this case , we observe that it yield same result as in fig 4.26

Now that we have computed the normalized attention weights, we are ready for the final step illustrated in figure 3.10: calculating the context vector z(2) by multiplying the embedded input tokens, x(i), with the corresponding attention weights and then summing the resulting vectors.

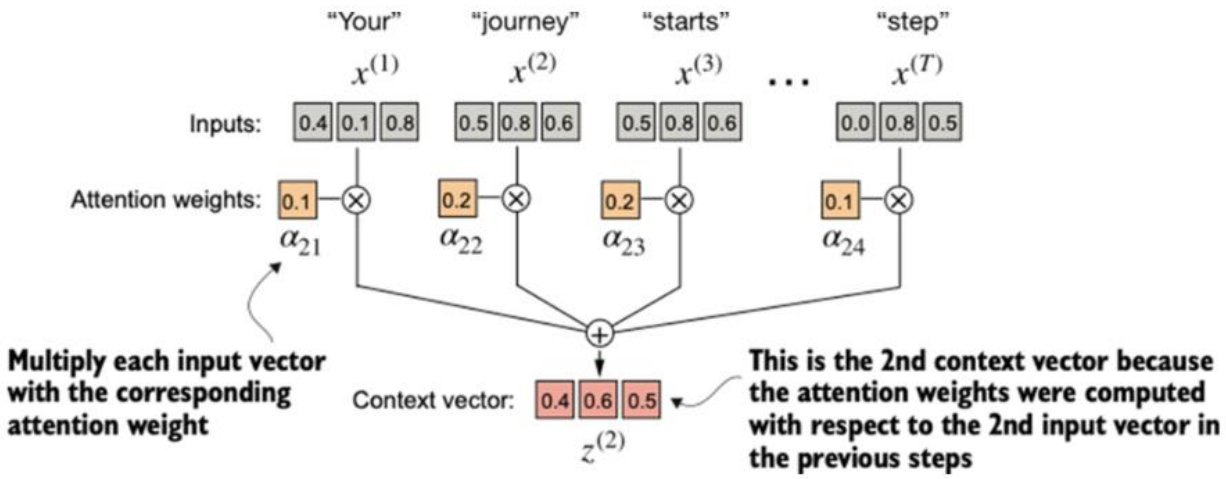


Figure 4.28: Computing Context Vectors

The context vector z(2) depicted in figure 3.10 is calculated as a weighted sum of all input vectors. This involves multiplying each input vector by its corresponding attention weight:

```
query = inputs[1] # 2nd input token is the query
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
```

Figure 4.29: Code for Context Vector

The results of this computation are as follows:

```
tensor([0.4419, 0.6515, 0.5683])
```

To compute attention weights for all input tokens we follow the same steps except that we make a few modifications in the code to compute all context vectors instead of only the second context vector, z(2).

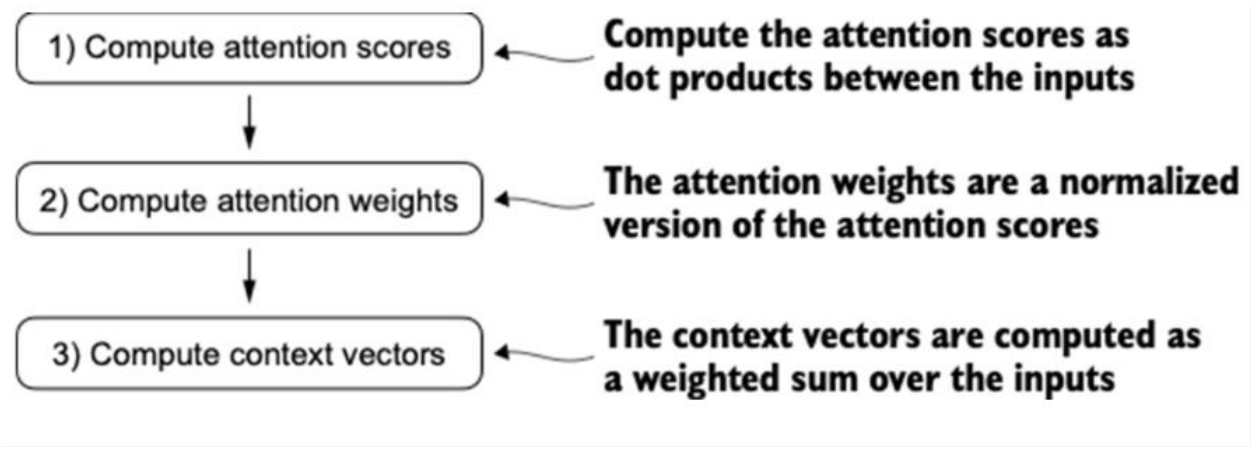


Figure 4.30: Cotext Vector Calculation Cycle

The self-attention mechanism with trainable weights builds on the previous concepts: we want to compute context vectors as weighted sums over the input vectors specific to a certain input element. The most notable difference is the introduction of weight matrices that are updated during model training. These trainable weight matrices are crucial so that the model (specifically, the attention module inside the model) can learn to produce "good" context vectors. We will implement the self-attention mechanism step by step by introducing the three trainable weight matrices Wq, Wk, and Wv. These three matrices are used to project the embedded input tokens, x(i), into query, key, and value vectors.

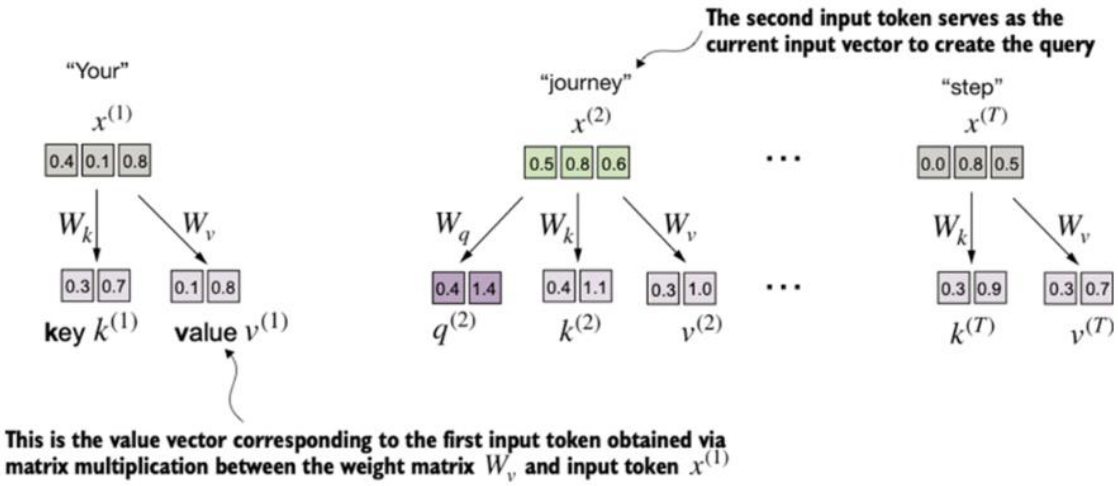


Figure 4.31: Steps in self-attention mechanism

We defined the second input element x(2) as the query when we computed the simplified attention weights to compute the context vector z(2). We generalized this to compute all context vectors z(1) ... z(T) for the six-word input sentence "Your journey starts with one step.". Even though our temporary goal is only to compute

the one context vector, z(2), we still require the key and value vectors for all input elements as they are involved in computing the attention weights with respect to the query q(2). We compute the attention weights by scaling the attention scores and using the softmax function we used earlier. The difference from earlier is that we now scale the attention scores by dividing them by the square root of the embedding dimension of the keys (note that taking the square root is mathematically the same as exponentiating by 0.5)

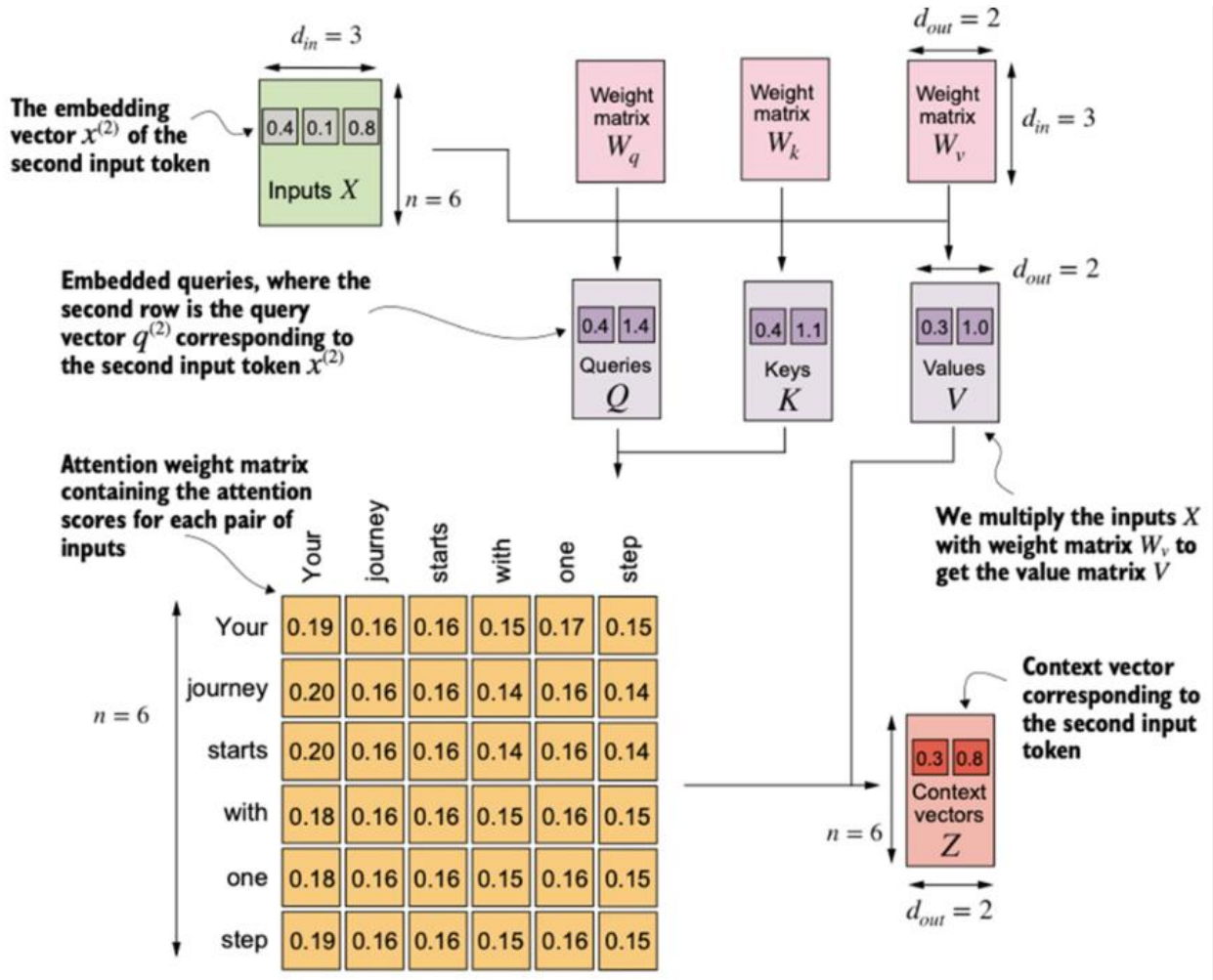


Figure 4.32: Computation of attention weight matrix based on resulting Queries(Q) and Keys(K)

Self-attention involves the trainable weight matrices Wq, Wk, and Wv. These matrices transform input data into queries, keys, and values, which are crucial components of the attention mechanism. As the model is exposed to more data during training, it adjusts these trainable weights, as we will see in upcoming chapters.

We will make enhancements to the self-attention mechanism, focusing specifically on incorporating causal and multi-head elements. The causal aspect involves modifying the attention mechanism to prevent the model from accessing future information in the sequence, which is crucial for tasks like language modeling, where each word prediction should only depend on previous words.

The multi-head component involves splitting the attention mechanism into multiple "heads." Each head learns different aspects of the data, allowing the model to simultaneously attend to information from different representation subspaces at different positions. This improves the model's performance in complex tasks.

Causal attention, also known as masked attention, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.

Consequently, when computing attention scores, the causal attention mechanism ensures that the model only factors in tokens that occur at or before the current token in the sequence.

To achieve this in GPT-like LLMs, for each token processed, we mask out the future tokens, which come after the current token in the input text.

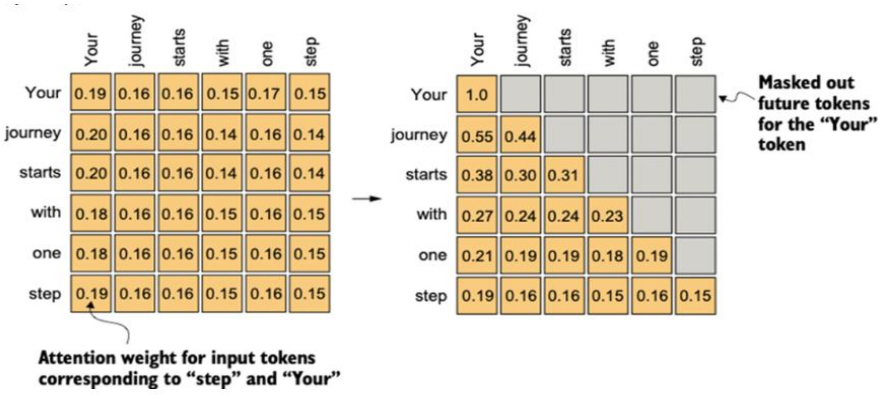


Figure 4.33: Masking out of attention weights in causal attention

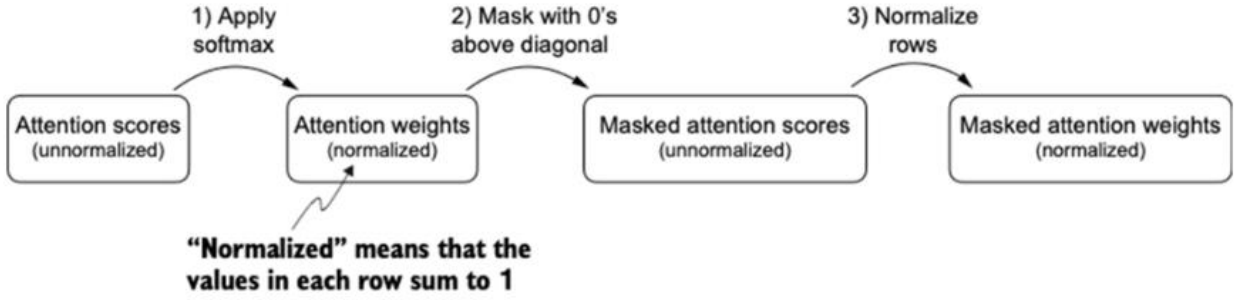


Figure 4.34: A more efficient way to obtain masked attention weight matrix

The softmax function converts its inputs into a probability distribution. When negative infinity values (-∞) are present in a row, the softmax function treats them as zero probability. (Mathematically, this is because e∞ approaches 0.)

We can implement this more efficient masking "trick" by creating a mask with 1s above the diagonal and then replacing these 1s with negative infinity (-inf) values

We could now use the modified attention weights to compute the context vectors via context_vec = attn_weights @ values.

Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied in two specific areas: after calculating the attention scores or after applying the attention weights to the value vectors.

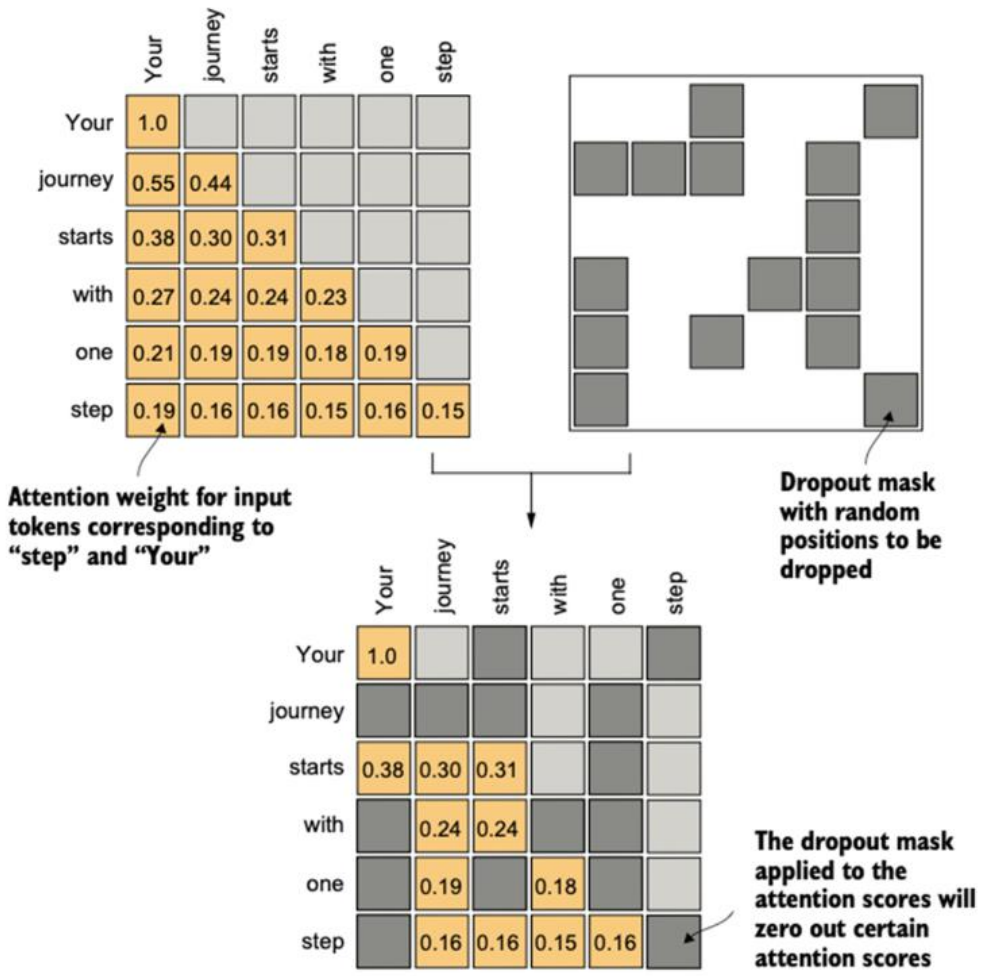


Figure 4.35: An additional droupout mask to reduce overfitting

When applying dropout to an attention weight matrix with a rate lets suppose 50%, half of the elements in the matrix are randomly set to zero. To compensate for the reduction in active elements, the values of the remaining elements in the matrix are scaled up by a factor of 1/0.5 = 2. This scaling is crucial to maintain the overall balance of the attention weights, ensuring that the average influence of the attention mechanism remains consistent during both the training and inference phases.

We now extend the previously implemented causal attention class over multiple heads. This is also called multi-head attention. The term "multi-head" refers to dividing the attention mechanism into multiple "heads," each operating independently. In this context, a single causal attention module can be considered single-head attention, where there is only one set of attention weights processing the input sequentially. , we will tackle this expansion from causal attention to multi-head attention. The first subsection will intuitively build a multi-head attention module by

stacking multiple CausalAttention modules for illustration purposes. The second subsection will then implement the same multi-head attention module in a more complicated but more computationally efficient way.

In practical terms, implementing multi-head attention involves creating multiple instances of the self-attention mechanism each with its own weights, and then combining their outputs. Using multiple instances of the self-attention mechanism can be computationally intensive, but it's crucial for the kind of complex pattern recognition that models like transformer-based LLMs are known for.

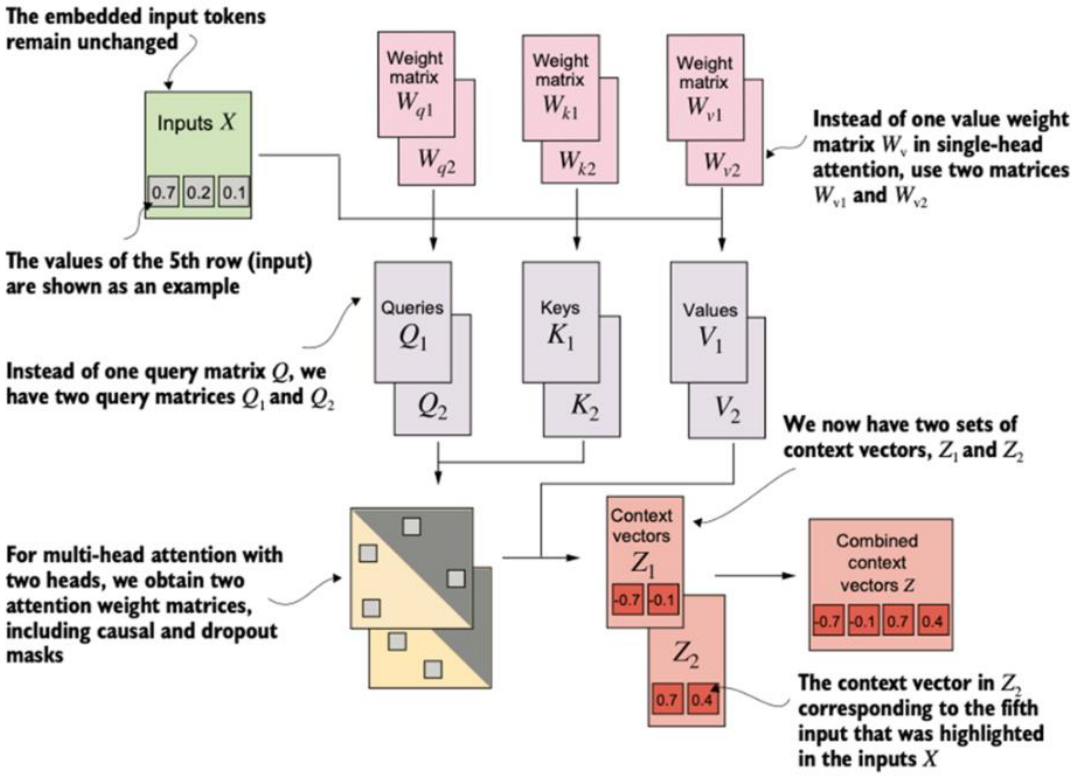


Figure 4.36: Multi-Head attention

The main idea behind multi-head attention is to run the attention mechanism multiple times (in parallel) with different, learned linear projections—the results of multiplying the input data (like the query, key, and value vectors in attention mechanisms) by a weight matrix.

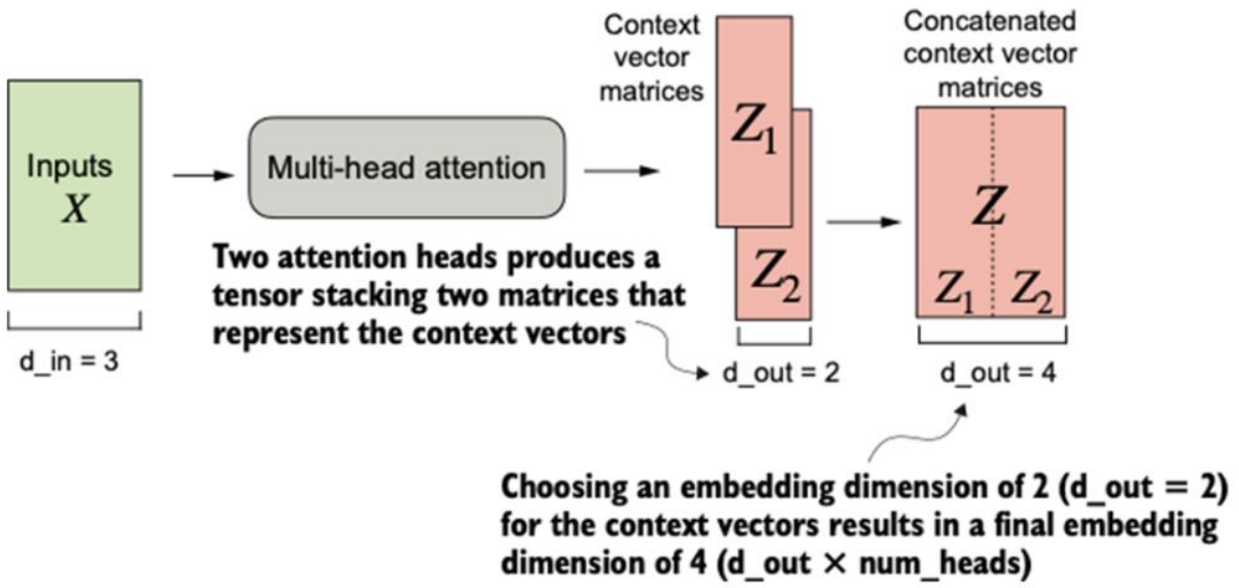


Figure 4.37 Multihead Attention with num_heads=2:

We can improve this implementation by processing the heads in parallel. One way to achieve this is by computing the outputs for all attention heads simultaneously via matrix multiplication, as we will explore in the next section.

The splitting of the query, key, and value tensors, as depicted in figure 3.26, is achieved through tensor reshaping and transposing operations using PyTorch's .view and .transpose methods. The input is first transformed (via linear layers for queries, keys, and values) and then reshaped to represent multiple heads.

The key operation is to split the d_out dimension into num_heads and head_dim, where head_dim = d_out / num_heads. This splitting is then achieved using the .view method: a tensor of dimensions (b, num_tokens, d_out) is reshaped to dimension (b, num_tokens, num_heads, head_dim).

The tensors are then transposed to bring the num_heads dimension before the num_tokens dimension, resulting in a shape of (b, num_heads, num_tokens, head_dim). This transposition is crucial for correctly aligning the queries, keys, and values across the different heads and performing batched matrix multiplications efficiently.

The matrix multiplication implementation in PyTorch handles the four-dimensional input tensor so that the matrix multiplication is carried out between the two last dimensions (num_tokens, head_dim) and then repeated for the individual heads.

For comparison, the smallest GPT-2 model (117 million parameters) has 12 attention heads and a context vector embedding size of 768. The largest GPT-2 model (1.5 billion parameters) has 25 attention heads and a context vector embedding size of 1,600. Note that the embedding sizes of the token inputs and context embeddings are the same in GPT models (d_in = d_out).

In the context of deep learning and LLMs like GPT, the term "parameters" refers to the trainable weights of the model. These weights are essentially the internal variables of the model that are adjusted and optimized during the training process to minimize a specific loss function. This optimization allows the model to learn from the training data.

For example, in a neural network layer that is represented by a 2,048 × 2,048-dimensional matrix (or tensor) of weights, each element of this matrix is a parameter. Since there are 2,048 rows and 2,048 columns, the total number of parameters in this layer is 2,048 multiplied by 2,048, which equals 4,194,304 parameters.

```
GPT_CONFIG_124M = {
    "vocab_size": 50257,  # Vocabulary size
    "context_length": 1024,      # Context length
    "emb_dim": 768,       # Embedding dimension
    "n_heads": 12,        # Number of attention heads
    "n_layers": 12,       # Number of layers
    "drop_rate": 0.1,     # Dropout rate
    "qkv_bias": False     # Query-Key-Value bias
}
```

Figure 4.38: config dictionary for GPT

In the GPT_CONFIG_124M dictionary, we use concise variable names for clarity and to prevent long lines of code:

- vocab_size refers to a vocabulary of 50,257 words, as used by the BPE tokenizer
- context_length denotes the maximum number of input tokens the model can handle via the positional embeddings
- emb_dim represents the embedding size, transforming each token into a 768-dimensional vector.
- n_heads indicates the count of attention heads in the multi-head attention mechanism
- n_layers specifies the number of transformer blocks in the model, which will be elaborated on in upcoming sections.

- drop_rate indicates the intensity of the dropout mechanism (0.1 implies a 10% drop of hidden units) to prevent overfitting
- qkv_bias determines whether to include a bias vector in the Linear layers of the multi-head attention for query, key, and value computations.

```python
import torch
import torch.nn as nn

class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg)
              for _ in range(cfg["n_layers"])]
        )
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x):
        return x

class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()

    def forward(self, x):
        return x
```

Figure 4.39: A dummy GPT Architexture class

The DummyGPTModel class in this code defines a simplified version of a GPT-like model using PyTorch's neural network module (nn.Module). The model architecture in the DummyGPTModel class consists of token and positional embeddings,

dropout, a series of transformer blocks (DummyTransformerBlock), a final layer normalization (DummyLayerNorm), and a linear output layer (out_head). The configuration is passed in via a Python dictionary, for instance, the GPT_CONFIG_124M dictionary we created earlier.

The forward method describes the data flow through the model: it computes token and positional embeddings for the input indices, applies dropout, processes the data through the transformer blocks, applies normalization, and finally produces logits with the linear output layer.

The preceding code is already functional, as we will see later in this section after we prepare the input data. However, for now, note in the preceding code that we have used placeholders (DummyLayerNorm and DummyTransformerBlock) for the transformer block and layer normalization.

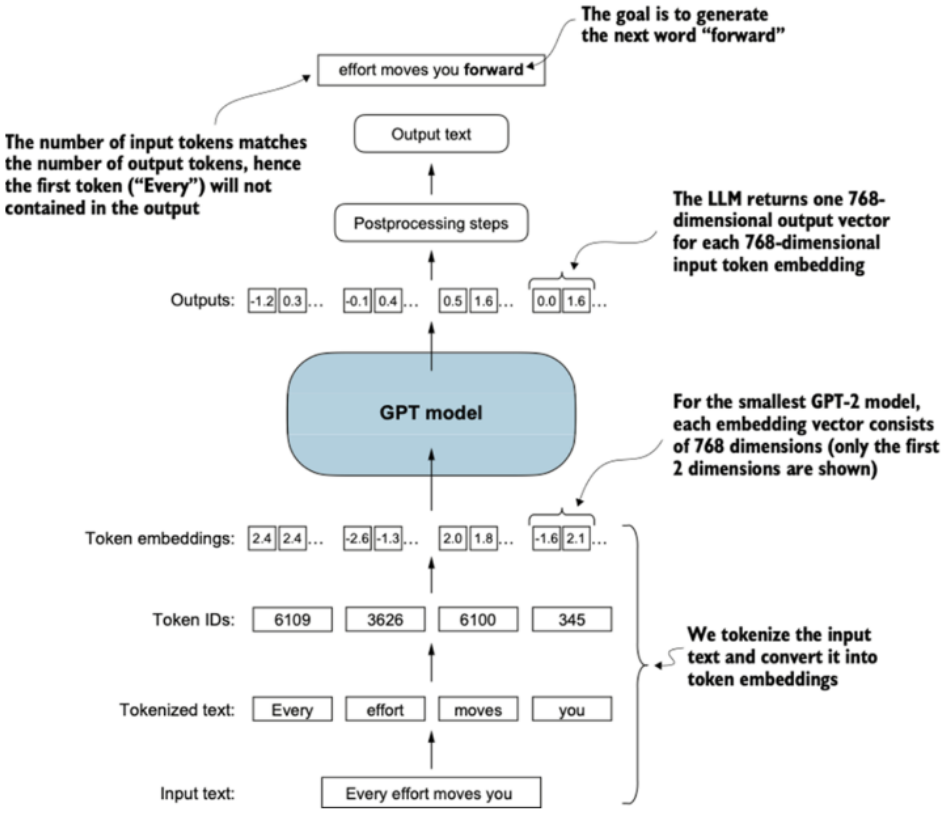


Figure 4.40: how the input data is tokenized, embedded, and fed to the GPT model

We tokenize a batch consisting of two text inputs for the GPT model using the tiktoken tokenizer

```
import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"

batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
```

Figure 4.41: Tokenizing using tiktoken

The resulting token IDs for the two texts are as follows:

```
tensor([[6109,  3626,  6100,   345],
        [6109,  1110,  6622,   257]])
```

Next, we initialize a new 124 million parameter DummyGPTModel instance and feed it the tokenized batch

```
torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

Figure 4.42: Initializing GPT class

The model outputs, which are commonly referred to as logits, are as follows:

```
Output shape: torch.Size([2, 4, 50257])
tensor([[[-1.2034,  0.3201, -0.7130,  ..., -1.5548, -0.2390, -0.4667],
         [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2430],
         [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5835],
         [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0400]],

        [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4530],
         [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6621],
         [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3717],
         [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1814]]],
       grad_fn=<UnsafeViewBackward0>)
```

Figure 4.43: logits

The output tensor has two rows corresponding to the two text samples. Each text sample consists of four tokens; each token is a 50,257-dimensional vector, which matches the size of the tokenizer's vocabulary.

The embedding has 50,257 dimensions because each of these dimensions refers to a unique token in the vocabulary. At the end of this chapter, when we implement the postprocessing code, we will convert these 50,257-dimensional vectors back into token IDs, which we can then decode into words.

Now that we have taken a top-down look at the GPT architecture and its in- and outputs, we will code the individual placeholders in the upcoming sections, starting with the real layer normalization class that will replace the DummyLayerNorm in the previous code.

Training deep neural networks with many layers can sometimes prove challenging due to problems like vanishing or exploding gradients. These problems lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function. In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions.

The main idea behind layer normalization is to adjust the activations (outputs) of a neural network layer to have a mean of 0 and a variance of 1, also known as unit variance. This adjustment speeds up the convergence to effective weights and ensures consistent, reliable training. As we have seen in the previous section, based on the DummyLayerNorm placeholder, in GPT-2 and modern transformer architectures, layer normalization is typically applied before and after the multi-head attention module and before the final output layer.

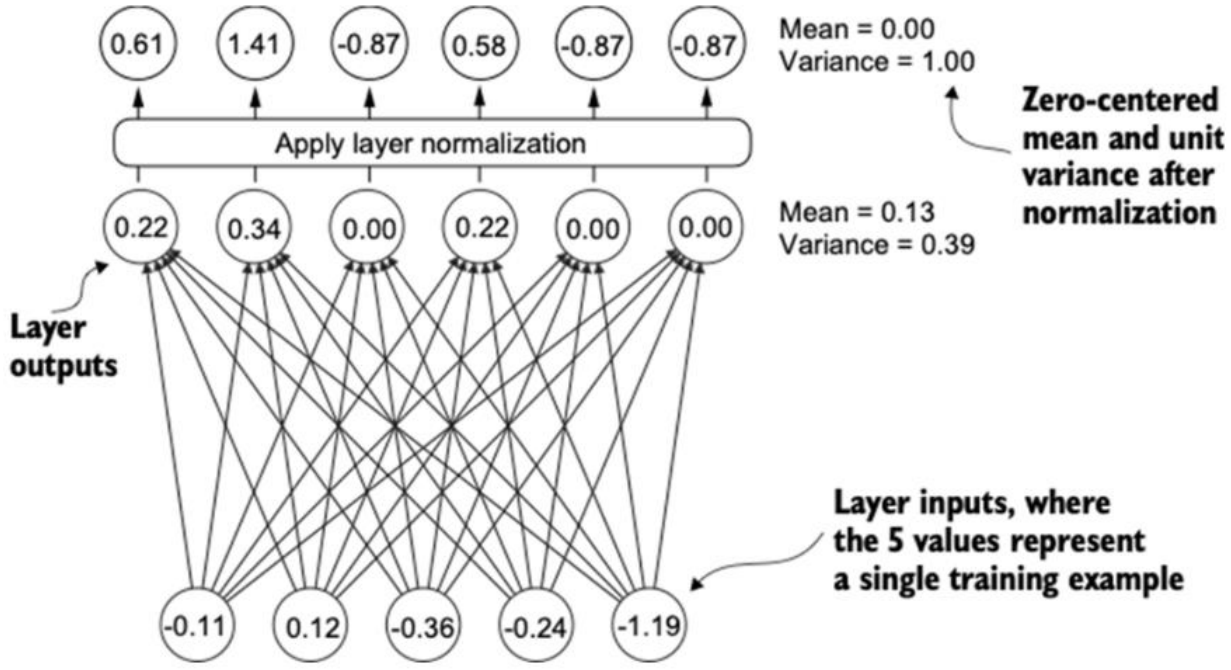


Figure 4.44: Layer Normalization

Historically, the ReLU activation function has been commonly used in deep learning due to its simplicity and effectiveness across various neural network architectures. However, in LLMs, several other activation functions are employed beyond the traditional ReLU. Two notable examples are GELU (Gaussian error linear unit) and SwiGLU (Swish-gated linear unit).

GELU and SwiGLU are more complex and smooth activation functions incorporating Gaussian and sigmoid-gated linear units, respectively. They offer improved performance for deep learning models, unlike the simpler ReLU.

The GELU activation function can be implemented in several ways; the exact version is defined as GELU(x) = x·Φ(x), where Φ(x) is the cumulative distribution function of the standard Gaussian distribution. In practice, however, it's common to implement a computationally cheaper approximation (the original GPT-2 model was also trained with this approximation):

$$GELU(x) \approx 0.5 \cdot x \cdot \left(1 + tanh\left[\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715 \cdot x^3\right)\right]\right)$$

Figure 4.45: GELU

ReLU is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero. GELU is a smooth, nonlinear function that approximates ReLU but with a non-zero gradient for negative values.

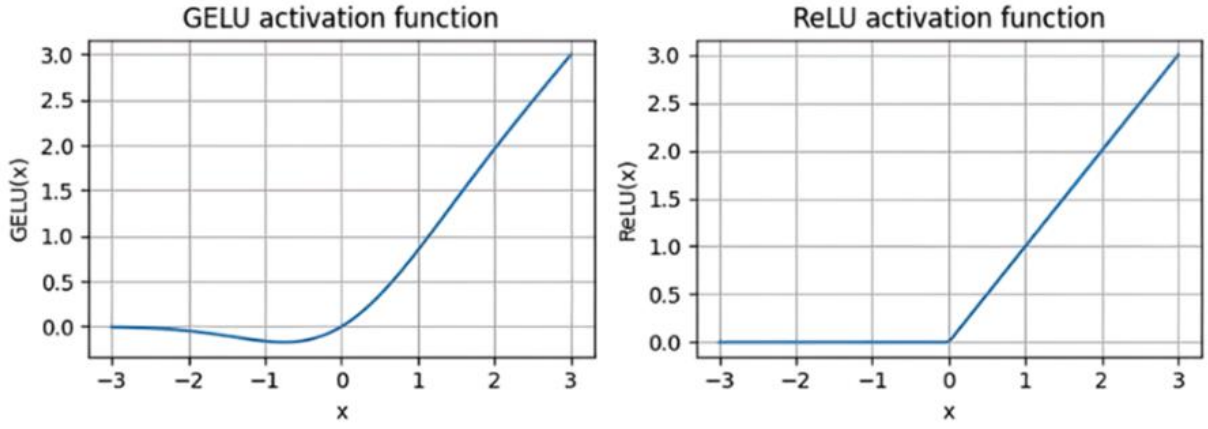


Figure 4.46: Outputs of GELU vs ReLU

The smoothness of GELU, can lead to better optimization properties during training, as it allows for more nuanced adjustments to the model's parameters. In contrast, ReLU has a sharp corner at zero, which can sometimes make optimization harder, especially in networks that are very deep or have complex architectures. Moreover, unlike RELU, which outputs zero for any negative input, GELU allows for a small, non-zero output for negative values. This characteristic means that during the training process, neurons that receive negative input can still contribute to the learning process, albeit to a lesser extent than positive inputs.

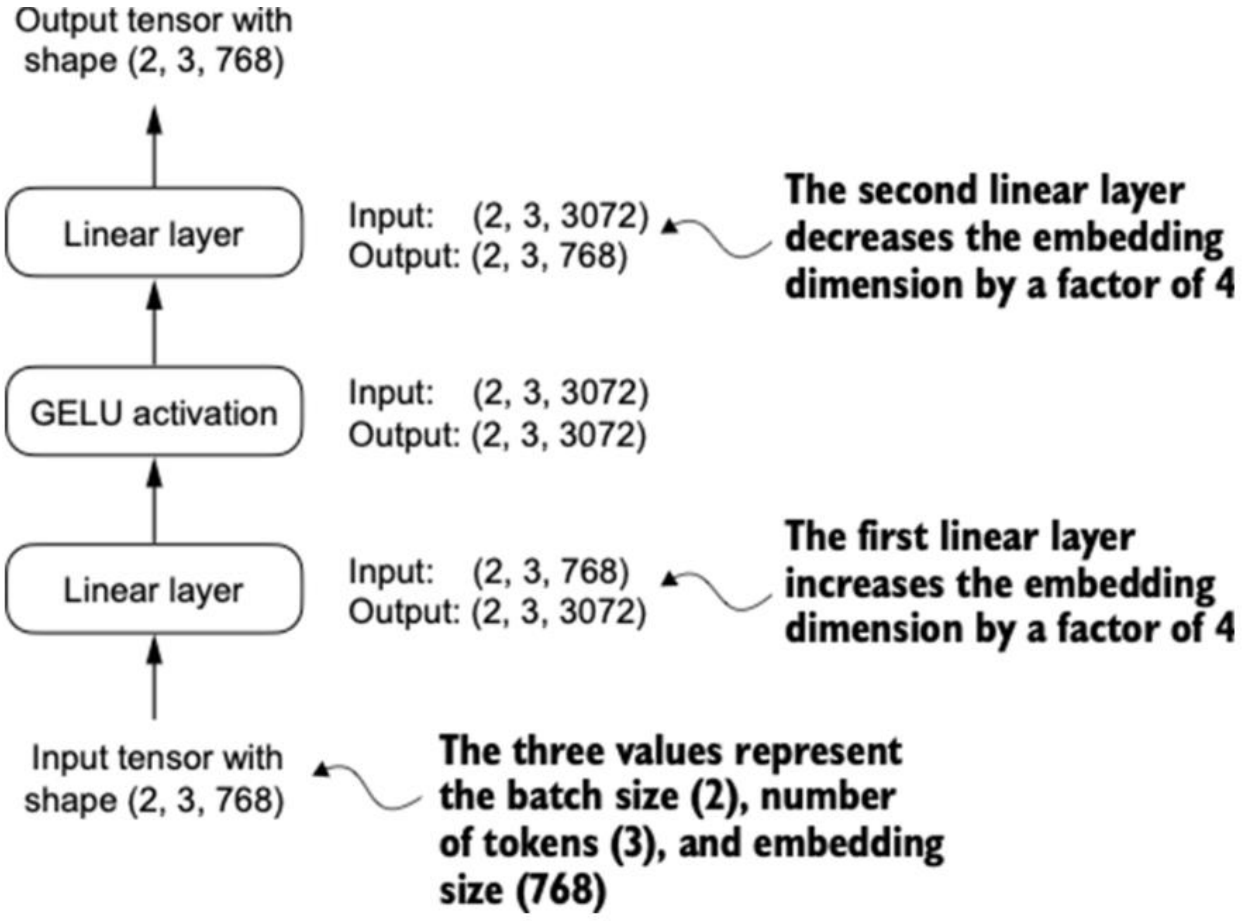


Figure 4.47: Connection between layers and feed forward neural network

The FeedForward module plays a crucial role in enhancing the model's ability to learn from and generalize the data. Although the input and output dimensions of this module are the same, it internally expands the embedding dimension into a higher-dimensional space through the first linear layer. This expansion is followed by a nonlinear GELU activation and then a contraction back to the original dimension

with the second linear transformation. Such a design allows for the exploration of a richer representation space.

Moreover, the uniformity in input and output dimensions simplifies the architecture by enabling the stacking of multiple layers, as we will do later, without the need to adjust dimensions between them, thus making the model more scalable.

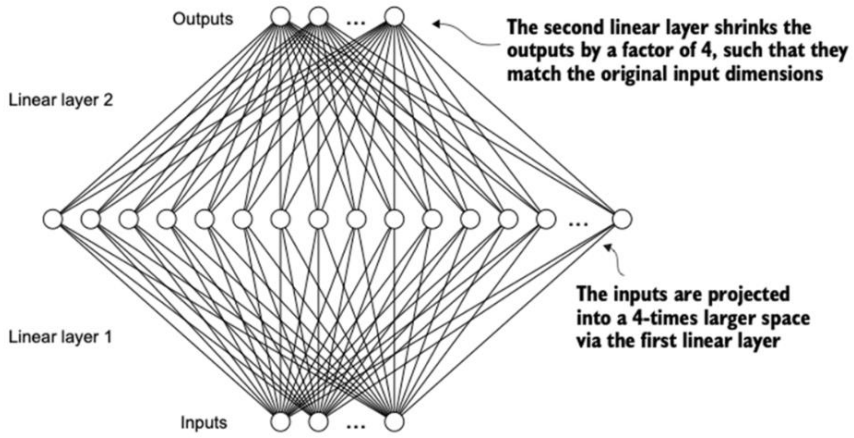


Figure 4.48: Expansion and Contraction of layer outputs

The concept behind shortcut connections, also known as skip or residual connections. Originally, shortcut connections were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients. The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers

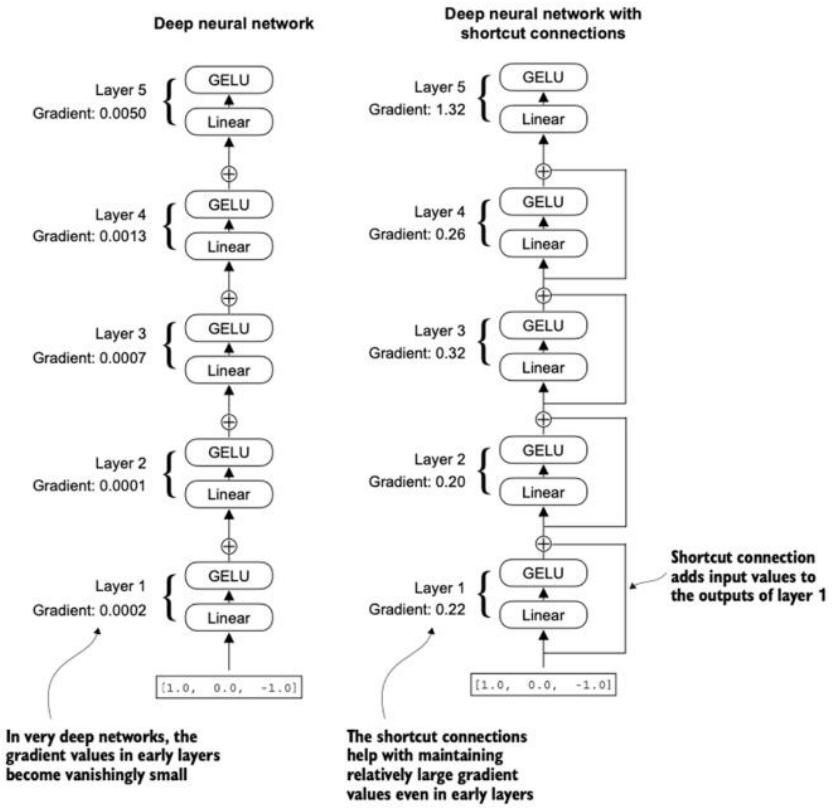


Figure 4.48: Comparison between architectures with and without shortcut connections

a shortcut connection creates an alternative, shorter path for the gradient to flow through the network by skipping one or more layers, which is achieved by adding the output of one layer to the output of a later layer. This is why these connections are also known as skip connections. They play a crucial role in preserving the flow of gradients during the backward pass in training

we specify a loss function that computes how close the model output and a user-specified target (here, for simplicity, the value 0) are. Then, when calling loss.backward(), PyTorch computes the loss gradient for each layer in the model. We can iterate through the weight parameters via model.named_parameters(). Suppose we have a 3 × 3 weight parameter matrix for a given layer.

In short, the .backward() method is a convenient method in PyTorch that computes loss gradients, which are required during model training, without implementing the math for the gradient calculation ourselves, thereby making working with deep neural networks much more accessible.

In conclusion, shortcut connections are important for overcoming the limitations posed by the vanishing gradient problem in deep neural networks. Shortcut connections are a core building block of very large models such as LLMs, and they will help facilitate more effective training by ensuring consistent gradient flow across layers when we train the GPT model

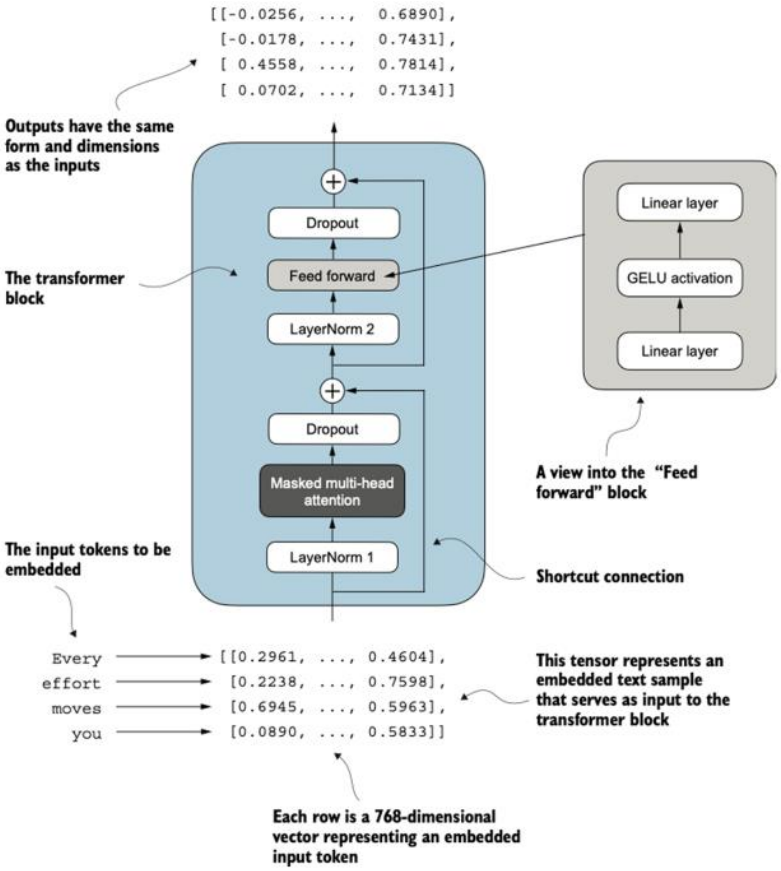


Figure 4.50: Transformer Block

When a transformer block processes an input sequence, each element in the sequence (for example, a word or subword token) is represented by a fixed-size vector (in the case of figure 4.13, 768 dimensions). The operations within the transformer block, including multi-head attention and feed forward layers, are designed to transform these vectors in a way that preserves their dimensionality.

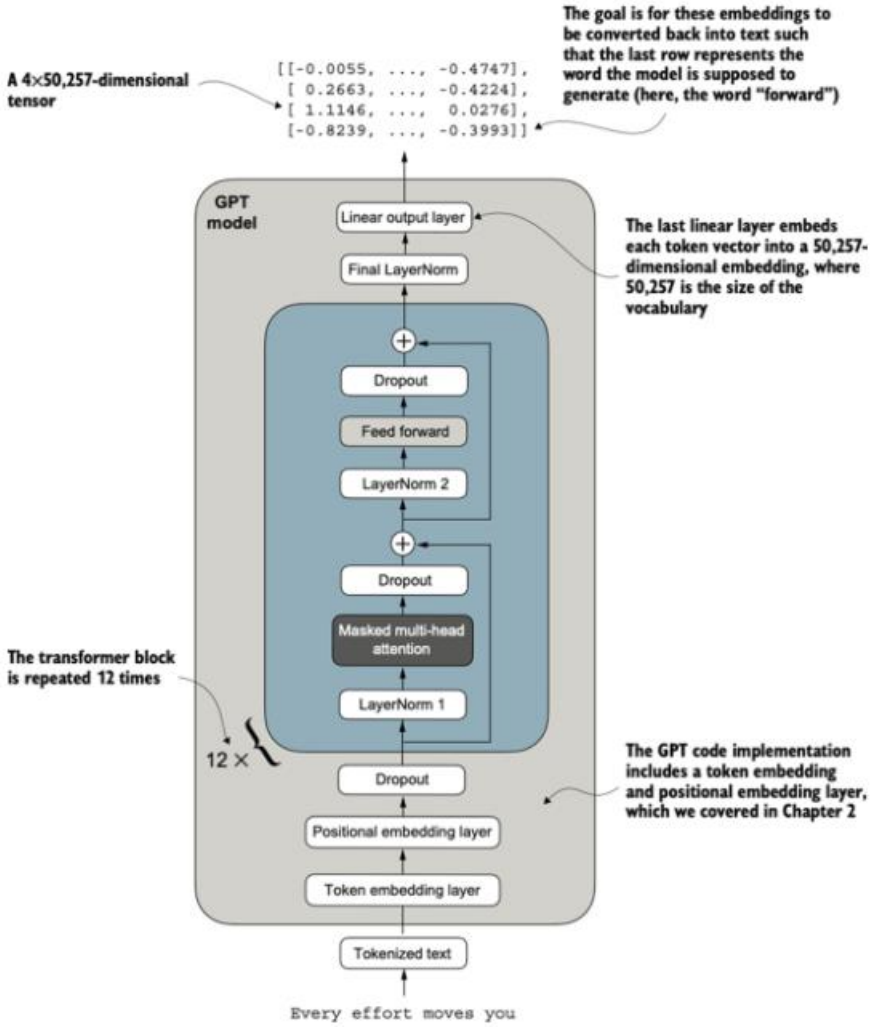
The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence. In contrast, the feed forward network modifies the data individually at each position. This combination not only enables a more nuanced understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.

Layer normalization (LayerNorm) is applied before each of these two components, and dropout is applied after them to regularize the model and prevent overfitting. This is also known as Pre-LayerNorm. Older architectures, such as the original transformer model, applied layer normalization after the self-attention and feed forward networks instead, known as Post-LayerNorm, which often leads to worse training dynamics.

The class also implements the forward pass, where each component is followed by a shortcut connection that adds the input of the block to its output. This critical feature helps gradients flow through the network during training and improves the learning of deep models

The transformer block maintains the input dimensions in its output, indicating that the transformer architecture processes sequences of data without altering their shape throughout the network.

The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design. This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship. However, the output is a context vector that encapsulates information from the entire input sequence, as we learned in chapter 3. This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.



Listing 4.7 The GPT model architecture implementation

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)

        pos_embeds = self.pos_emb(
            torch.arange(seq_len, device=in_idx.device)
        )
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Figure 4.51: GPT Model Architecture and Code

In the case of the 124 million parameter GPT-2 model, it's repeated 12 times, which we specify via the n_layers entry in the GPT_CONFIG_124M dictionary. In the case of the largest GPT-2 model with 1,542 million parameters, this transformer block is repeated 36 times.

As shown in figure 4.51, the output from the final transformer block then goes through a final layer normalization step before reaching the linear output layer. This layer maps the transformer's output to a high-dimensional space (in this case, 50,257 dimensions, corresponding to the model's vocabulary size) to predict the next token in the sequence.

The __init__ constructor of this GPTModel class initializes the token and positional embedding layers using the configurations passed in via a Python dictionary, cfg. These embedding layers are responsible for converting input token indices into dense vectors and adding positional information, as discussed in chapter 2.

Next, the __init__ method creates a sequential stack of TransformerBlock modules equal to the number of layers specified in cfg. Following the transformer blocks, a LayerNorm layer is applied, standardizing the outputs from the transformer blocks to stabilize the learning process. Finally, a linear output head without bias is defined, which projects the transformer's output into the vocabulary space of the tokenizer to generate logits for each token in the vocabulary.

The forward method takes a batch of input token indices, computes their embeddings, applies the positional embeddings, passes the sequence through the transformer blocks, normalizes the final output, and then computes the logits, representing the next token's unnormalized probabilities. We will convert these logits into tokens and text outputs in the next section.

Let's now initialize the 124 million parameter GPT model using the GPT_CONFIG_124M dictionary we pass into the cfg parameter and feed it with the batch text input

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)
```

Figure 4.52: Initializing GPT Model with config (fig 4.38)

The preceding code prints the contents of the input batch followed by the output tensor:

```
Input batch:
 tensor([[6109,  3626,  6100,   345], # token IDs of text 1
         [6109,  1110,  6622,   257]]) # token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613,  0.4222, -0.0711,  ...,  0.3483,  0.4661, -0.2838],
         [-0.1792, -0.5660, -0.9485,  ...,  0.0477,  0.5181, -0.3168],
         [ 0.7120,  0.0332,  0.1085,  ...,  0.1018, -0.4327, -0.2553],
         [-1.0076,  0.3418, -0.1190,  ...,  0.7195,  0.4023,  0.0532]],

        [[-0.2564,  0.0900,  0.0335,  ...,  0.2659,  0.4454, -0.6806],
         [ 0.1230,  0.3653, -0.2074,  ...,  0.7705,  0.2710,  0.2246],
         [ 1.0558,  1.0318, -0.2800,  ...,  0.6936,  0.3205, -0.3178],
         [-0.1565,  0.3926,  0.3288,  ...,  1.2630, -0.1858,  0.0388]]],
       grad_fn=<UnsafeViewBackward0>)
```

Figure 4.53: Output Tensor by GPT code(fig 4.52)

the output tensor has the shape [2, 4, 50257], since we passed in two input texts with four tokens each. The last dimension, 50,257, corresponds to the vocabulary size of the tokenizer. In the next section, we will see how to convert each of these 50,257-dimensional output vectors back into tokens.

Before we move on to the next section and code the function that converts the model outputs into text, let's spend a bit more time with the model architecture itself and analyze its size.

Weight tying that is used in the original GPT-2 architecture, which means that the original GPT-2 architecture is reusing the weights from the token embedding layer in its output layer. Weight tying reduces the overall memory footprint and computational complexity of the model. However, in my experience, using separate token embedding and output layers results in better training and model performance; hence, we use separate layers in our GPTModel implementation. The same is true for modern LLMs.

by calculating the memory requirements for the 163 million parameters in our GPTModel object and assuming each parameter is a 32-bit float taking up 4 bytes, we find that the total size of the model amounts to 621.83 MB, illustrating the relatively large storage capacity required to accommodate even relatively small LLMs.

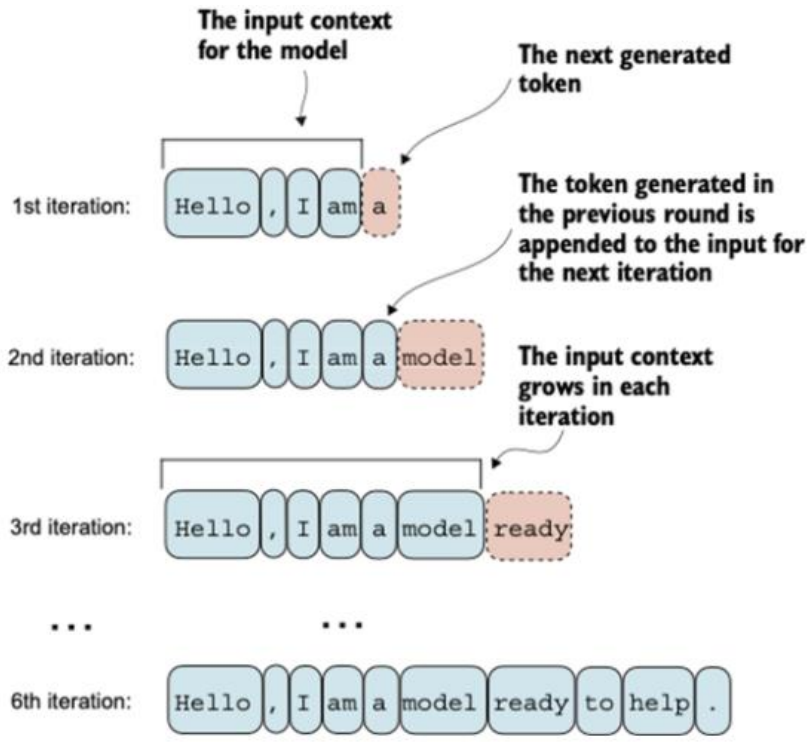


Figure 4.54: Step-by-step process by which LLMs generate text

The above figure illustrates the step-by-step process by which a GPT model generates text given an input context, such as "Hello, I am," on a big-picture level. With each iteration, the input context grows, allowing the model to generate coherent and contextually appropriate text. By the sixth iteration, the model has constructed a complete sentence: "Hello, I am a model ready to help."

In the previous section, we saw that our current GPTModel implementation outputs tensors with shape [batch_size, num_token, vocab_size]

The process by which a GPT model goes from output tensors to generated text involves several steps, as illustrated in the below figure. These steps include decoding the output tensors, selecting tokens based on a probability distribution, and converting these tokens into human-readable text.

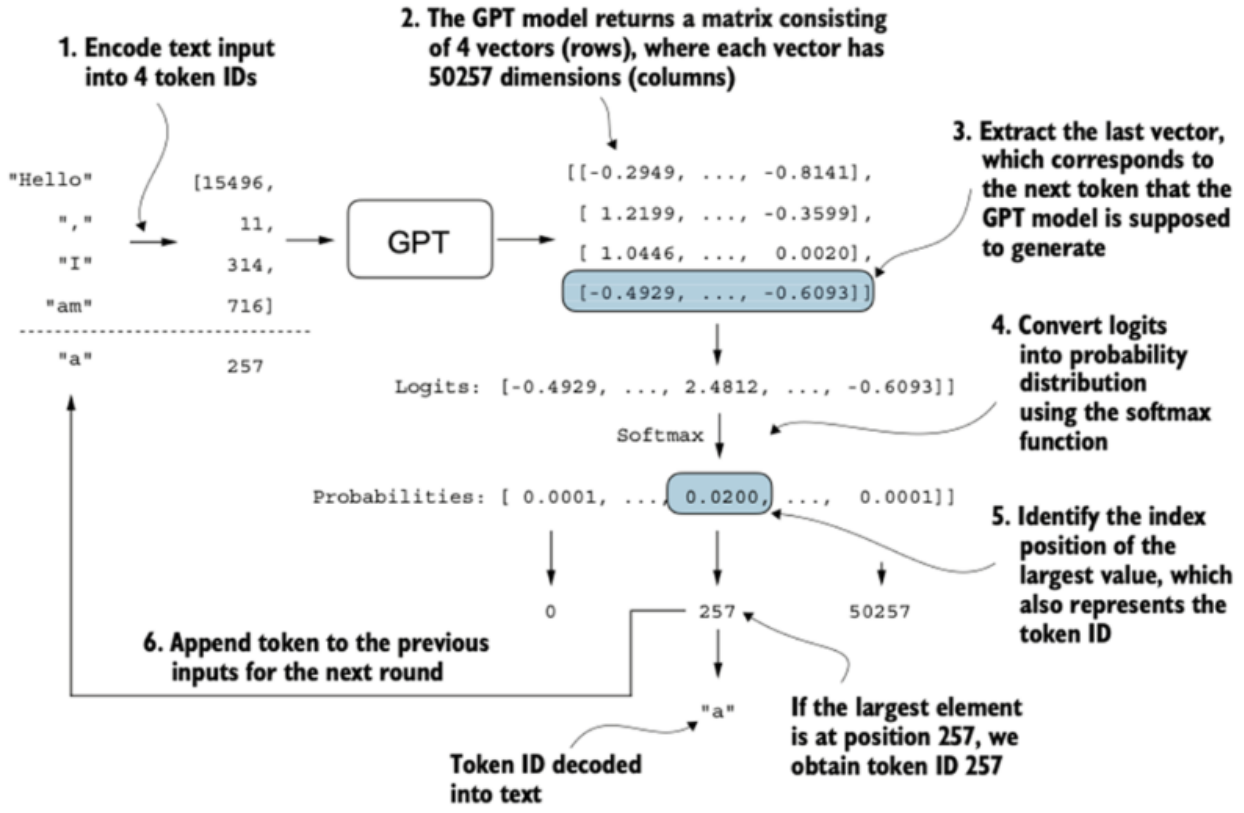


Figure 4.55: Single iteration in GPT

The next-token generation process detailed illustrates a single step where the GPT model generates the next token given its input. In each step, the model outputs a matrix with vectors representing potential next tokens. The vector corresponding to the next token is extracted and converted into a probability distribution via the softmax function. Within the vector containing the resulting probability scores, the index of the highest value is located, which translates to the token ID. This token ID is then decoded back into text, producing the next token in the sequence. Finally, this token is appended to the previous inputs, forming a new input sequence for the subsequent iteration. This step-by-step process enables the model to generate text sequentially, building coherent phrases and sentences from the initial input context.

In practice, we repeat this process over many iterations, until we reach a user-specified number of generated tokens. In code, we can implement the token-generation process as shown in the following listing.

Listing 4.8 A function for the GPT model to generate text

```
def generate_text_simple(model, idx,
                         max_new_tokens, context_size):
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)

        logits = logits[:, -1, :]
        probas = torch.softmax(logits, dim=-1)
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Figure 4.56: Text generation function

The code snippet provided demonstrates a simple implementation of a generative loop for a language model using PyTorch. It iterates for a specified number of new tokens to be generated, crops the current context to fit the model's maximum context size, computes predictions, and then selects the next token based on the highest probability prediction.

In the preceding code, the generate_text_simple function, we use a softmax function to convert the logits into a probability distribution from which we identify the position with the highest value via torch.argmax. The softmax function is monotonic, meaning it preserves the order of its inputs when transformed into outputs. So, in practice, the softmax step is redundant since the position with the highest score in the softmax output tensor is the same position in the logit tensor. In other words, we could apply the torch.argmax function to the logits tensor directly and get identical results. However, we coded the conversion to illustrate the full process of transforming logits to probabilities, which can add additional intuition so that the model generates the most likely next token, which is known as greedy decoding.

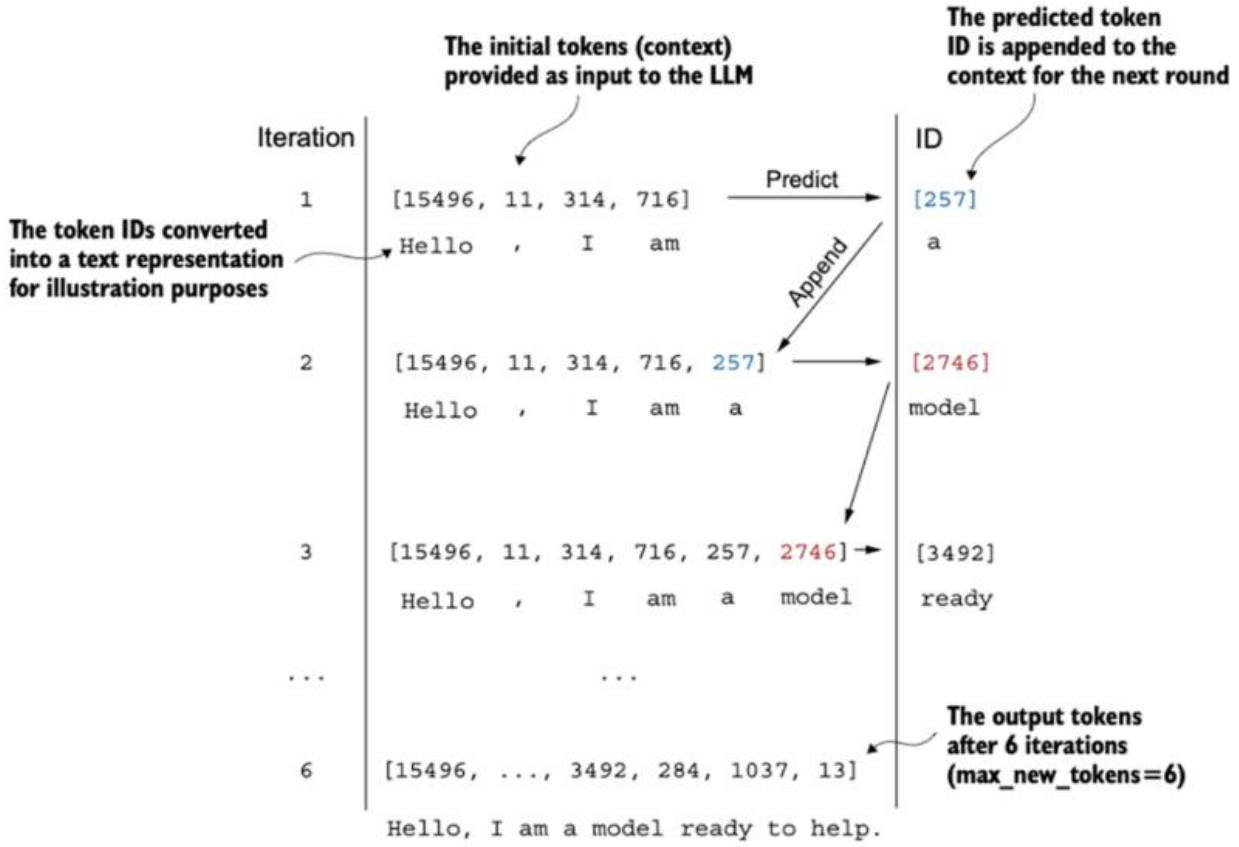


Figure 4.57: Six iterations of token prediction cycle

we generate the token IDs in an iterative fashion. For instance, in iteration 1, the model is provided with the tokens corresponding to "Hello, I am," predicts the next token (with ID 257, which is "a"), and appends it to the input. This process is repeated until the model produces the complete sentence "Hello, I am a model ready to help" after six iterations.

So basically, the above steps summarize how a LLM is built in general, after a general architecture is coded we pass unlabeled data into the architecture in a loop as illustrated in the below image

It outlines eight steps, starting with iterating over each epoch, processing batches, resetting gradients, calculating the loss and new gradients, and updating weights and concluding with monitoring steps like printing losses and generating text samples.

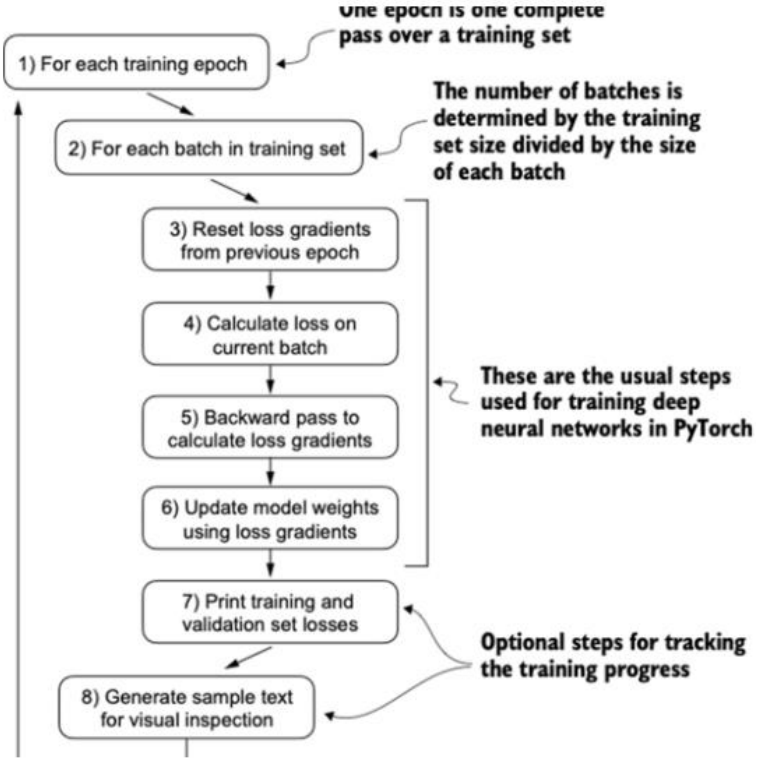


Figure 4.58: Typical training loop

In code, we can implement this training flow via the train_model_simple function in the following listing.

```
def train_model_simple(model, train_loader, val_loader,
                       optimizer, device, num_epochs,
                        eval_freq, eval_iter, start_context, tokenizer):
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            loss = calc_loss_batch(
                input_batch, target_batch, model, device
            )
            loss.backward()
            optimizer.step()
            tokens_seen += input_batch.numel()
            global_step += 1

            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        generate_and_print_sample(
            model, tokenizer, device, start_context
        )
    return train_losses, val_losses, track_tokens_seen
```

Figure 4.59: Function for pretraining LLMs

The evaluate_model function corresponds to step 7 in figure 5.11. It prints the training and validation set losses after each model update so we can evaluate whether the training improves the model.

More specifically, the evaluate_model function calculates the loss over the training and validation set while ensuring the model is in evaluation mode with gradient tracking and dropout disabled when calculating the loss over the training and validation sets:

```
1  def evaluate_model(model, train_loader, val_loader, device, eval_iter):
2      model.eval()
3      with torch.no_grad():
4          train_loss = calc_loss_loader(
5              train_loader, model, device, num_batches=eval_iter
6          )
7          val_loss = calc_loss_loader(
8              val_loader, model, device, num_batches=eval_iter
9          )
10     model.train()
11     return train_loss, val_loss
```

Figure 4.60: evaluate model code

Similar to evaluate_model, the generate_and_print_sample function is a convenience function that we use to track whether the model improves during the

training. In particular, the generate_and_print_sample function takes a text snippet (start_context) as input, converts it into token IDs, and feeds it to the LLM to generate a text sample using the generate_text_simple function we used earlier:

```
def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
        decoded_text = token_ids_to_text(token_ids, tokenizer)
        print(decoded_text.replace("\n", " "))  # Compact print format
    model.train()
```

Figure 4.61: Improvement tracking function

While the evaluate_model function gives us a numeric estimate of the model's training progress, this generate_and_print_sample text function provides a concrete text example generated by the model to judge its capabilities during training.

Let's see this all in action by training a GPTModel instance for 10 epochs using an AdamW optimizer and the train_model_simple function we defined earlier:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(
     model.parameters(),
    lr=0.0004, weight_decay=0.1
)
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=1,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

Figure 4.62: Training GPT Model using AdamW Optimizer

Executing the training_model_simple function starts the training process, which takes about 5 minutes on a MacBook Air or a similar laptop to complete. The output printed during this execution is as follows:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and,
 and, and, and, and, and, and, and, and, and, and, and,, and, and,
[...] #A Results are truncated to save space
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?"  "Yes--quite insensible to the irony. She wanted
him vindicated--and by me!"  He laughed again, and threw back the
window-curtains, I had the donkey. "There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know," was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed luncheon-table, when, on a
later day, I had again run over from Monte Carlo; and Mrs. Gis
```

Figure 4.63: Code output of fig 4.62

As we can see, based on the results printed during the training, the training loss improves drastically, starting with a value of 9.558 and converging to 0.762. The language skills of the model have improved quite a lot. In the beginning, the model is only able to append commas to the start context (Every effort moves you,,,,,,,,,,,,) or repeat the word and. At the end of the training, it can generate grammatically correct text.

Similar to the training set loss, we can see that the validation loss starts high (9.856) and decreases during the training. However, it never becomes as small as the training set loss and remains at 6.372 after the 10th epoch.

Before discussing the validation loss in more detail, let's create a simple plot that shows the training and validation set losses side by side:

```python
import matplotlib.pyplot as plt
def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(
        epochs_seen, val_losses, linestyle="-.", label="Validation loss"
    )
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax2 = ax1.twiny()
    ax2.plot(tokens_seen, train_losses, alpha=0)
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```

Figure 4.64: Function to plot graph

The resulting training and validation loss plot is shown in the below figure

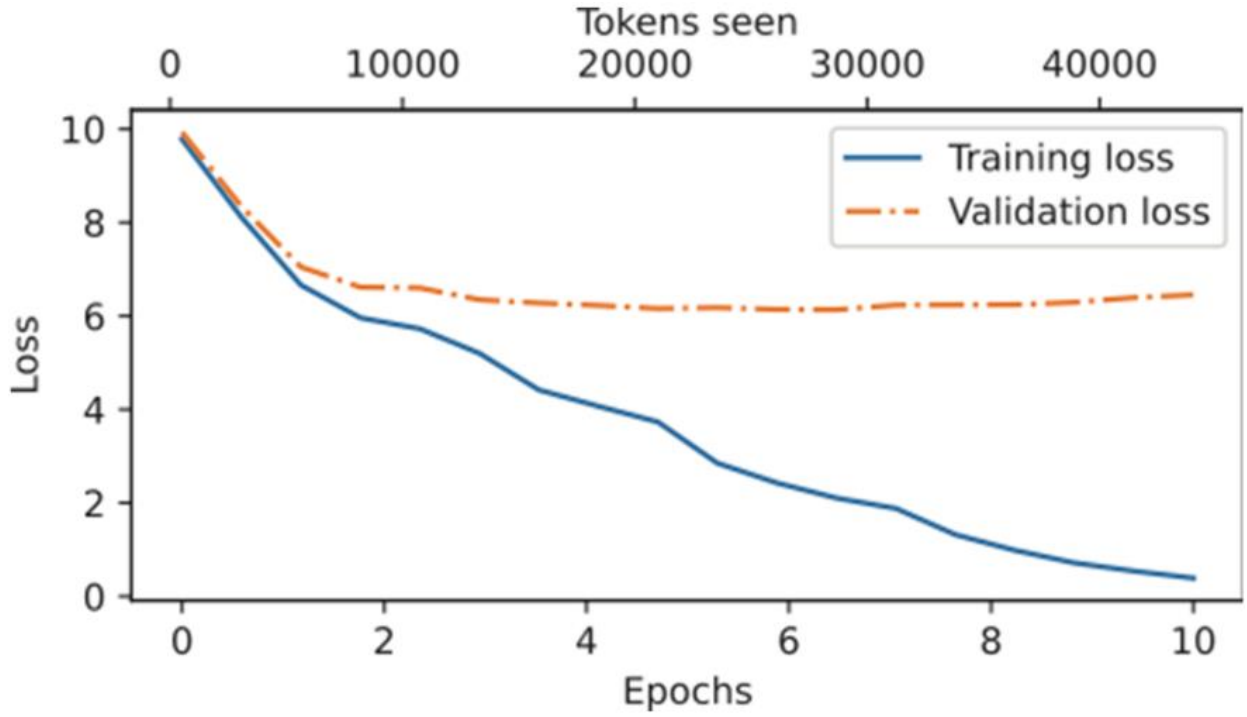


Figure 4.65: Training vs Validation loss

both the training and validation losses start to improve for the first epoch. However, the losses start to diverge past the second epoch. This divergence and the fact that the validation loss is much larger than the training loss indicate that the model is overfitting to the training data. We can confirm that the model memorizes the training data verbatim by searching for the generated text snippets, such as quite insensible to the irony in the "The Verdict" text file.

This memorization is expected since we are working with a very, very small training dataset and training the model for multiple epochs. Usually, it's common to train a model on a much larger dataset for only one epoch. As mentioned earlier, interested readers can try to train the model on 60,000 public domain books from Project Gutenberg, where this overfitting does not occur.

In conclusion, building large language models is a complex endeavor that combines sophisticated neural networks, vast amounts of data, and immense computational power. By ingesting text and learning patterns from these datasets, LLMs are able to statistically predict the next word in a sequence, allowing them to generate human-

like text, translate languages, write different kinds of creative content, and answer your questions in an informative way.

Given the substantial resources required to train large language models, this study will leverage Gemma 2b, a powerful yet computationally efficient LLM. While other models might offer greater capabilities, Gemma 2b strikes a balance between performance and resource utilization, making it a suitable choice for our exploration.

We propose to use a pretrained Small Language model by cutting down its vocabulary size. By focusing the model on a relevant subset of tokens, decoder trimming reduces the chance of it encountering unexpected vocabulary during inference. This targeted approach can help mitigate errors and hallucinations by keeping the model's predictions grounded in the specific vocabulary it has been trained on for the task at hand. In essence, by reducing the model's decision-making space, decoder trimming promotes more reliable and accurate outputs.

For this study we use Google's Gemma series of models. It comes in two sizes: 2B and 7B parameters. Each size is available with a base (pre-trained) and instruction-tuned version. These are text-to-text, decoder-only large language models, available in English, with open weights. We use the Gemma-2B variant as our base model. The Gemma 2B model is quite intriguing due to its compact size. It has 2 billion parameters, which ensures that it has a small footprint. It takes only 2Gb of Storage and 5.1Gb of VRAM to load the model.

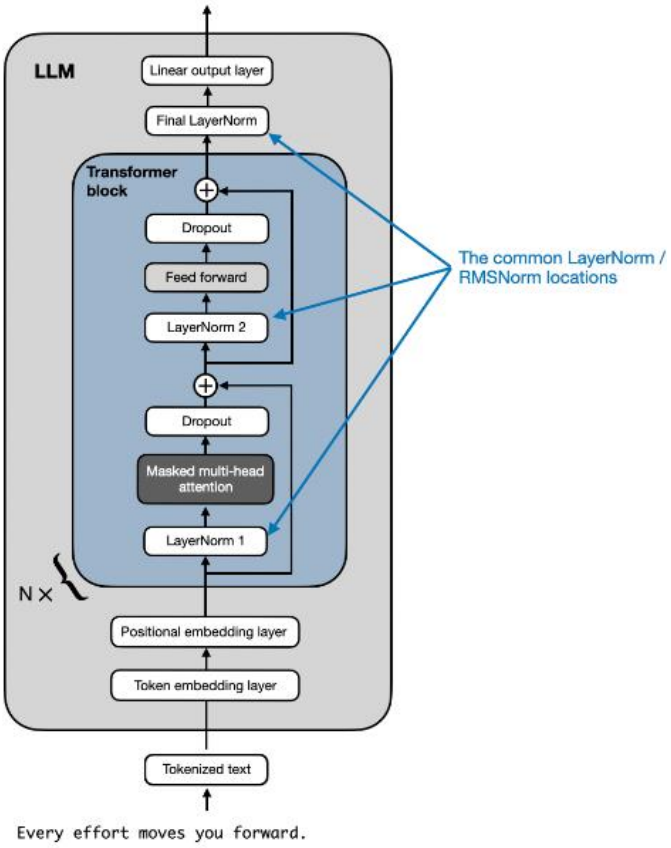


Figure 4.66: Gemma 2B Architecture (similar to GPT, Llama2)

Based on the original transformer decoder architecture (From "Attention Is All You Need" paper) with the below improvements:

Multi-Query Attention instead of the original multi-head attention.

RoPE Embeddings in each layer, sharing them across inputs and outputs to reduce model size.

GeGLU Activations instead of ReLU

Normalizer Location: Normalizes both input and output of each transformer sub-layer using RMSNorm.

Gemma 2b has 524,550,144 Embedding Parameters and 1,981,884,416 Non-Embedding Parameters

Parameters	2B
*d*_model	2048
Layers	18
Feedforward hidden dims	32768
Num heads	8
Num KV heads	1
Head size	256
Vocab size	256128

Figure 4.67: Key Model Parameters

In the original model, the final layer responsible for generating token predictions projects the hidden states to the vocabulary size. This layer's dimensions are typically defined by the model's hidden size and the total number of tokens in its vocabulary. For instance, if the hidden size is 768 and the vocabulary size is 256,000, the layer's dimensions would be (768, 256,000). This setup means that each token in the vocabulary corresponds to an embedding vector of size 768.

```
GemmaForCausalLM(
  (model): GemmaModel(
    (embed_tokens): Embedding(256000, 2048, padding_idx=0)
    (layers): ModuleList(
      (0-17): 18 x GemmaDecoderLayer(
        (self_attn): GemmaSdpaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=256, bias=False)
          (v_proj): Linear(in_features=2048, out_features=256, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (rotary_emb): GemmaRotaryEmbedding()
        )
        (mlp): GemmaMLP(
          (gate_proj): Linear(in_features=2048, out_features=16384, bias=False)
          (up_proj): Linear(in_features=2048, out_features=16384, bias=False)
          (down_proj): Linear(in_features=16384, out_features=2048, bias=False)
          (act_fn): PytorchGELUTanh()
        )
        (input_layernorm): GemmaRMSNorm()
        (post_attention_layernorm): GemmaRMSNorm()
      )
    )
    (norm): GemmaRMSNorm()
  )
  (lm_head): Linear(in_features=2048, out_features=256000, bias=False)
)
```

Figure 4.68: Raw Gemma 2b with 256k vocab size

When we want to trim the model to use only a subset of tokens, we create a new projection layer with the same input size but a reduced output size, reflecting the smaller vocabulary subset. For example, if we want to use a subset of 8 tokens, the new layer would have dimensions (768, 8), effectively reducing the vocabulary size from 256,000 to 8.

To ensure the new layer has the correct weights for these tokens, we copy the relevant weights from the original layer. We map each original token ID to a new token ID, copying the corresponding weight vector to the new position. This process extracts the necessary embedding vectors for the subset of tokens and discards the rest.

By replacing the original projection layer with this new, trimmed layer, the model will only produce outputs for the specified subset of tokens. This reduces computational overhead and tailors the model to the specific tokens of interest. Finally, verifying the dimensions of the new layer and the total number of parameters confirms that the trimming process was successful. This method allows the model to focus on a specific set of tokens, making it more efficient for targeted use cases.

```
GemmaForCausalLM(
  (model): GemmaModel(
    (embed_tokens): Embedding(256000, 2048, padding_idx=0)
    (layers): ModuleList(
      (0-17): 18 x GemmaDecoderLayer(
        (self_attn): GemmaSdpaAttention(
          (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (k_proj): Linear(in_features=2048, out_features=256, bias=False)
          (v_proj): Linear(in_features=2048, out_features=256, bias=False)
          (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
          (rotary_emb): GemmaRotaryEmbedding()
        )
        (mlp): GemmaMLP(
          (gate_proj): Linear(in_features=2048, out_features=16384, bias=False)
          (up_proj): Linear(in_features=2048, out_features=16384, bias=False)
          (down_proj): Linear(in_features=16384, out_features=2048, bias=False)
          (act_fn): PytorchGELUTanh()
        )
        (input_layernorm): GemmaRMSNorm()
        (post_attention_layernorm): GemmaRMSNorm()
      )
    )
    (norm): GemmaRMSNorm()
  )
  (lm_head): Linear(in_features=2048, out_features=8, bias=True)
)
```

Figure 4.30: Gemma 2b trimmed with only 8 tokens in vocabulary

Now with the decoder trimmed, we can write an efficient prompt to execute tasks according to the use-case.

For example, we can use a prompt like “Classify the following user input into one of positive or negative sentiment” with trimming the decoder layer with only “Positive” and “Negative” tokens for sentiment analysis use cases.

We can use a prompt like “Identify the intent of incoming email by its summary and classify it as Personal, Work, Events, Updates or Todos” to classify mails into categories by trimming the decoder output tokens to desired categories.

There are many use cases for this method like Spam Filtering, Topic Labelling, Binary Question Answering, Aspect-based Sentiment Analysis etc.

5. RESULTS AND DISCUSSION

Our analysis of two separate datasets yielded interesting insights with our proposed performing at par with state-of-the-art text classification. The first dataset, focusing on topic classification of news reports (AGNews), we achieved an accuracy score 90.6%. The following figure shows the Classification Report generated after running it on the test dataset.

	precision	recall	f1-score	support
1	0.92	0.89	0.90	1900
2	0.95	0.97	0.96	1900
3	0.85	0.87	0.86	1900
4	0.87	0.87	0.87	1900
accuracy			0.90	7600
macro avg	0.90	0.90	0.90	7600
weighted avg	0.90	0.90	0.90	7600

Figure 5.1: Classification Report on AGNews

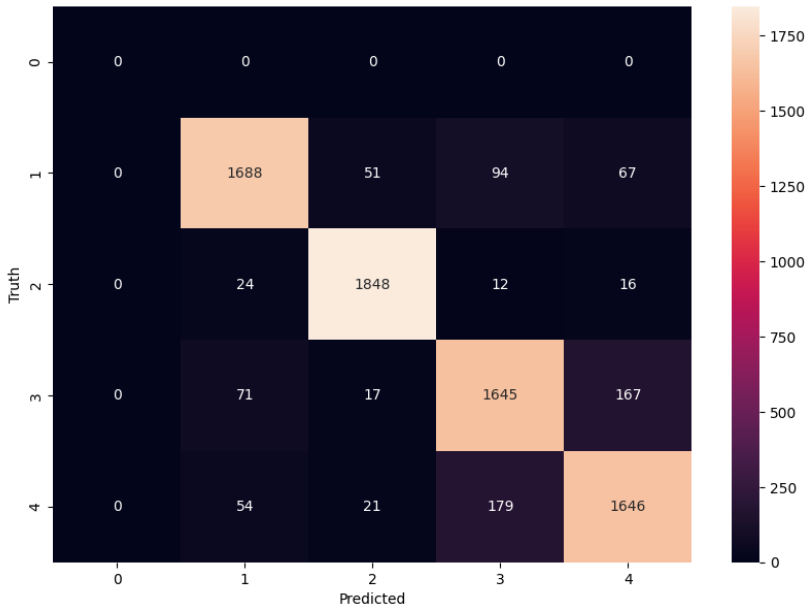


Figure 5.2: Confusion Matrix of AGNews

Model	*Accuracy Score(n=16)*
FT RoBERTa	*21.87*
GPT-3 Vanilla	89.47
GPT-3 Zero-shot-CoT	89.66
GPT-3 CRAP	90.16
Gemma-2b-Trimmed (Proposed Model)	**90.60**

Table 5.1: Comparison with SOTA Models on AGNews

On the second dataset in which we had to classify sentiments of the sentences (SST-2), we achieved an accuracy of 94%. The following figure shows the Classification Report generated after running it on the test dataset.

	precision	recall	f1-score	support
0	0.94	0.94	0.94	912
1	0.94	0.94	0.94	909
accuracy			0.94	1821
macro avg	0.94	0.94	0.94	1821
weighted avg	0.94	0.94	0.94	1821

Figure 5.3: Classification Report on SST-2

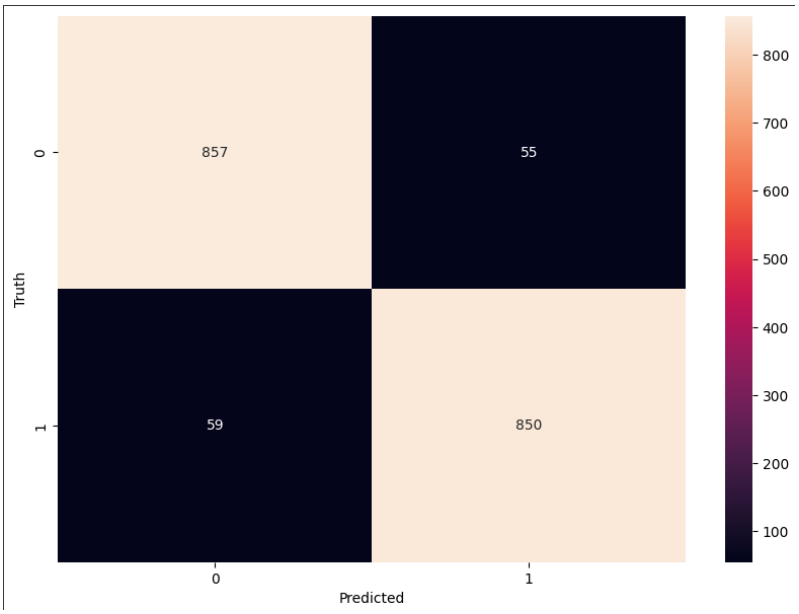


Figure 5.4: Confusion Matrix of SST-2

Model	Accuracy Score(n=16)
FT RoBERTa	*51.52*
GPT-3 Vanilla	90.15
GPT-3 Zero-shot-CoT	89.66
GPT-3 CRAP	90.48
Gemma-2b-Trimmed (Proposed Model)	**94.13**

Table 5.2: Comparison with SOTA models on SST-2

These findings suggest that decoder trimming presents a compelling and practical alternative for text classification tasks leveraging LLMs. It offers advantages in efficiency and potentially reduces reliance on large amounts of labeled data, making LLMs more accessible for broader adoption.

The promising results of decoder trimming on text classification tasks open exciting avenues for further exploration. Here are some key areas for future research:

- **LLM Architecture Compatibility:** This study focused on the Gemma-2B variant model. Future work can investigate the applicability of decoder trimming to a wider range of LLM architectures, including different encoder-decoder structures and parameter sizes. Evaluating performance across diverse architectures will provide a more comprehensive understanding of decoder trimming's generalizability.
- **Beyond Text Classification:** LLMs are powerful tools for various NLP tasks. Investigating the effectiveness of decoder trimming on tasks like sentiment analysis, question answering, and text summarization can broaden its impact within the NLP domain.
- **Dynamic Vocabulary Selection:** Currently, decoder trimming focuses on a static subset of relevant tokens. Exploring methods for dynamically selecting the vocabulary based on the specific classification task could further improve efficiency and accuracy.
- **Integration with Existing Training Pipelines:** Developing seamless integration of decoder trimming into existing fine-tuning pipelines would promote wider adoption. This could involve creating user-friendly tools or libraries that allow researchers and practitioners to easily leverage decoder trimming within their workflows.
- **Impact on Training Data Efficiency:** The initial findings suggest decoder trimming might require less training data compared to fine-tuning. Further

research should quantify this data efficiency advantage and explore its implications for resource-constrained scenarios.

- **Interpretability and Explainability:** Understanding how decoder trimming influences the decision-making process of LLMs is crucial. Developing methods to interpret and explain the model's reasoning behind classifications will enhance trust and facilitate debugging in case of errors.

By delving deeper into these areas, researchers can solidify decoder trimming as a powerful and efficient alternative for leveraging the capabilities of LLMs in various NLP tasks. This will ultimately lead to a more accessible and effective application of LLMs across a broader range of real-world use cases.